DEFENSE TECHNICAL INFORMATION CENTER

Information for the Defense Community

DTIC® has determined on 20/SEP/2010 that this Technical Document has the Distribution Statement checked below. The current distribution for this document can be found in the DTIC® Technical Report Database.

☒ **DISTRIBUTION STATEMENT A.** Approved for public release; distribution is unlimited.

☐ **© COPYRIGHTED**; U.S. Government or Federal Rights License. All other rights and uses except those permitted by copyright law are reserved by the copyright owner.

☐ **DISTRIBUTION STATEMENT B.** Distribution authorized to U.S. Government agencies only (fill in reason) (date of determination). Other requests for this document shall be referred to (insert controlling DoD office)

☐ **DISTRIBUTION STATEMENT C.** Distribution authorized to U.S. Government Agencies and their contractors (fill in reason) (date of determination). Other requests for this document shall be referred to (insert controlling DoD office)

☐ **DISTRIBUTION STATEMENT D.** Distribution authorized to the Department of Defense and U.S. DoD contractors only (fill in reason) (date of determination). Other requests shall be referred to (insert controlling DoD office).

☐ **DISTRIBUTION STATEMENT E.** Distribution authorized to DoD Components only (fill in reason) (date of determination). Other requests shall be referred to (insert controlling DoD office).

☐ **DISTRIBUTION STATEMENT F.** Further dissemination only as directed by (inserting controlling DoD office) (date of determination) or higher DoD authority.

Distribution Statement F is also used when a document does not contain a distribution statement and no distribution statement can be determined.

☐ **DISTRIBUTION STATEMENT X.** Distribution authorized to U.S. Government Agencies and private individuals or enterprises eligible to obtain export-controlled technical data in accordance with DoDD 5230.25; (date of determination). DoD Controlling Office is (insert controlling DoD office).

REPORT DOCUMENTATION PAGE

AFRL-SR-AR-TR-10-0299

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
02-26-2009	Final	06-01-2005 - 11-30-08

4. TITLE AND SUBTITLE
Security Certification Modeling

5a. CONTRACT NUMBER
FA9550-05-1-0374

5b. GRANT NUMBER
FA9550-05-1-0374

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Rose Gamble

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

University of Tulsa
800 S. Tucker Drive
Tulsa, OK 74145

8. PERFORMING ORGANIZATION REPORT NUMBER
Technical Report TR-SEAT-UTULSA-2009-4

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Office
of Scientific Research
875 North Randolph Street
Suite 325, Rm 3112
Arlington, VA 22203
RSL

10. SPONSOR/MONITOR'S ACRONYM(S)
AFOSR

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT
Open

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This research focused on security certification policy modeling for a System of Systems (SoS). Three main results were obtained. The first major result was a semi-formal UML Component Protection Profile (CPP)to describe a software component's broad security expectations and interactions within the SoS. The CPP allows direct comparison of components that interact to determine if they interfere with local security requirements. Examples illustrate basic instantiations of multiple component security profiles along with the local violations that result from their conflicting or competing interactions within a SoS. The second result was an extension to a formal specification language to accommodate SoS global architecture and security certification criteria expressed as progress properties. Audit criteria from the NIST SP800-53 exemplify both local and global constraints and their compliance throughout the SoS. The third major result is a formal analysis of role-based access control policies using an extension of the Colored Petri Net. Overall, this fundamental effort indicated that more unification of security constructs is needed across the local, global, and internal activities of a SoS and its components to determine full system compliance with security certification criteria.

15. SUBJECT TERMS
Security, Certification, System of Systems, Distributed Systems

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT unclassified	b. ABSTRACT unclassified	c. THIS PAGE unclassified	none	42	Rose Gamble
					19b. TELEPHONE NUMBER *(include area code)* 918-631-2988

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

Executive Summary

Due to constantly changing demands on today's systems, using component-based software systems is a popular development strategy for large applications. Legacy and stovepipe systems, commercial-off-the-shelf (COTS) products, and other third-party software created independently to interact with one another, are now being forced to work together reliably and securely as *components* of a System of Systems (SoS). When components become part of a SoS, individual component security is not enough to guarantee that the whole system is secure.

Software security certification must show compliance with a range of documented requirements associated with different security policies and targeted to certain computing and interface properties. The focus of this research is on investigating and reformulating those security certification criteria that impact SoS. Specifically, we are concerned with inter-component interaction behaviors that are governed by their individual security policies, as well and their communication style to exchange data and control information. The goal is to investigate modeling approaches so that the security policies within and across the SoS architecture could be expressed in such a way that compliance with certification criteria could be assessed.

The major challenges are

- Understanding policy attributes and mechanisms with respect to compliance
- Interleaving component communication styles and policy governance
- Devising common properties across different policy types
- Expressing security certification criteria within a comparative framework of the policies
- Verifying compliance based on both safety and progress properties
- Understanding the impacts of policies are internal component processing, component interfaces, and global SoS architecture

This report describes the culmination of our research to address the above areas. We partition the discussion into three dimensions of software security certification. We present the Security Certification Modeling (SCM) framework, a modeling profile that is based on a component's security policies, communication strategies, and placement in an SoS architecture. Security certification criteria based on audit requirements are used to show how local comparators are expressed to find policy violations among interacting components in the SoS. We overview our research effort toward increasing the formal representation of policy attributes at the global SoS level. We extend an existing formal specification language to create X-UNITY to express the SoS hierarchy, communication infrastructure, and component policies that affect SoS security requirements. X-UNITY allows for temporally based certification criteria to be formulated for compliance verification across the SoS components. Finally, we report on our early investigation into specific access control representations to understand how the SCM can be more detailed to express internal processing of security policy functions, such as access control. From this research, a unique Petri Net, called ConPN, has been developed to analyze violations of Role-Based Access Control policies of interacting components.

Publications related to the grant

- Isolation in Design Reuse, with M. Gamble, *Journal of Software Process Improvement and Practice*, Vol. 13, pp. 145-156, Feb. 2008.
- Reasoning about Hybrid System of Systems Designs, with M.T. Gamble, *IEEE International Conference on Composition Based Software Systems*, 2008.
- Isolating Mechanisms in COTS-Based Systems, with M.T. Gamble, *IEEE International Conference on COTS-Based Software Systems*, 2007.
- The impact of certification criteria on integrated COTS-based systems, with M. Kelkar, R. Perry, and A. Walvekar, Proceedings of the *IEEE International Conference on COTS-Based Software Systems*, 2007.
- Patterns of Conflict among Software Components, with M. Hepner, M. Kelkar, L. Davis, and D. Flagg, *Journal of Systems and Software*, Vol 79, Issue 4, pp. 537-551, 2006.
- Determining Conflicts in Interdomain Mappings for Access Control, with A. Walvekar, M. Kelkar, M. Smith, *Joint Workshop on Foundations of Computer Security and Automated Reasoning For Security Protocol Analysis*, (FCS-ARSPA'06), 2006.
- Examining Security Certification and Access Control Conflicts Using Deontic Logic, with M. Smith and M. Kelkar, *1st International Workshop on Software Certification* (CERTSOFT'06), 2006.
- Forming A Security Certification Enclave For Service-Oriented Architectures, with M. Hepner and M. T. Gamble, *Modeling, Design, and Analysis For Service-Oriented Architecture Workshop* (MDA4SOA'06), 2006.
- Interaction Partnering Criteria for Commercial-Off-The-Shelf (COTS) Components, with M. Kelkar, and M. Smith, *18th International Conference on Software Engineering and Knowledge Engineering, 2006* (SEKE 2006).
- Security Profiling of COTS Components For Interaction Partnering, with M. Kelkar and M. Smith, *IEEE International Conference on COTS-Based Software Systems, Poster Session*, Feb. 2006.
- Interdomain Policy and Access Control within A Grid Architecture, with N. Lamonica, M. Kelkar, and R. Baird, *Software Engineering and Application Conference*, 2005.

Contributing Theses and Dissertations

- Toward Policy Based Logic For Secure Component Interaction, M. Smith, M.S. Thesis, *Dept. of Mathematical and Computer Sciences, University of Tulsa*, 2006.
- ConPN: Detecting Conflicts in Interdomain Mappings, A. Walvekar, M.S. Thesis, *Dept. of Mathematical and Computer Sciences, University of Tulsa*, 2006.
- Modeling Software Component Security Policies, Ph.D. Dissertation, M. Kelkar, *Dept. of Mathematical and Computer Sciences, University of Tulsa*, 2007.
- Reasoning about Isolation in Distribute System Design, Ph.D. Dissertation, M.T. Gamble, *Dept. of Mathematical and Computer Sciences, University of Tulsa*, 2007.

1. Security Certification Modeling Framework

The Security Certification Modeling (SCM) framework focuses on a subset of available security policy types relevant to SoS. Emphasis is placed on policy properties for which communication and interaction is predictable and observable. This emphasis is derived from the need for the SoS to describe the mechanisms that implement expressed security policies via the attributes and functionality that relate specifically to interaction. Similarly, our policy investigation examines certification criteria that have a direct relevance to SoS exposed interfaces and services.

Security policies for SoSs rely on an communication-centric approach that maintains the *integrity* of system components, protects the *confidentiality* of information, and provides continued *availability* to systems [1, 2]. We derive the set of policy types from industry standards including the Department of Defense Goal Security Architecture (DGSA) [3], the Common Criteria [4], Carnegie Mellon University's Computer Emergency Response Team (CERT) [2], the Federal Information Processing Standards (FIPS) minimum security requirements [5], and the NIST 800 series of publications [6]. An empirical study of these documents reveals significant corroboration of the policy types across the documents. The DGSA defines seven core policy types [3]. The Common Criteria contains eleven functional classes for security policies [4]. NIST isolates security requirements across seventeen different categories [5].

Overlap exists in the policy designations. For instance, the DGSA policy type of *availability* [3] is closely related to the FIPS document definition of *contingency planning* [5] and the Common Criteria's *resource utilization* [4]. Selecting the appropriate consolidation of properties is required to express important policies with the details needed for analysis. From empirical analysis, we delineate features, properties, and intent across a uniform granularity to obtain a minimal set of core policy types for our focus. The result is six policy types and their relevant properties governing SoS interaction for preserving system integrity, confidentiality, and availability. The specific subset of policy types are: *audit, authentication, authorization, contingency planning, data protection,* and *non-repudiation*. Our SCM framework is designed to accommodate statements from each of these policy types for SoS component specification and interaction. We target the policy properties of interest to a SoS for each policy type below.

Audit is concerned with recording or logging specific events that occur within or between components of a SoS. Policy statements of interest express constraints on the type of information to be recorded, the logging frequency, and how the information is handled or aggregated [2, 4, 7]. Mechanisms associated with auditing focus on the creation of audit records and the manipulation of log files [2, 8]. Policy requirements can stipulate that audit records of specific levels of importance (normal, alert, catastrophic, etc.) require the transmission of messages (e-mail, log files, etc.) to administrative personnel or automated toolkits [9].

Component authentication policies concern the type and configuration of the authentication checking mechanisms for SoS components. Common authentication mechanisms that are used to verify component identity are password-based or certificate-based. Constraints may be placed on length, complexity, and validity periods [2, 4, 9]. Authentication policies have policy statements related to a Boolean response of authentication (allow, deny) [4]. Authentication may allow single login or interactive access for multiple logins [10]. Common policies restrict authentication based on

component location, express requirements over re-authentication after periods of inactivity, and restrict the number of failed login attempts that are permitted by the system [2].

Authorization policies state how authorization checks are performed by components within the SoS. Specific mechanisms for access control (lists, matrices, etc.) must be detailed as well as how access control is granted (mandatory access control, role-based access control, etc.) [10-12]. Policy specifications may require local authorization or enable remote authorization for the establishment of trusted channels for which information may be passed through [4]. Other policies track access time to establish temporal constraints for authorization [4, 11, 13].

Backup procedures and redundant system requirements are specified with contingency planning policy statements. Operational specifications of the contingency plan dictate what mechanisms should be available for expressing how backups should record data (differential, incremental, etc.) as well as any transactional database requirements that are used by the SoS (roll-back, roll-forward, etc.) [14, 15]. Where and how the backup systems are stored within the SoS, such as initiating a dedicated component or tasking an existing component, are specified with contingency planning policy statements. Thus, the policy can define local or remote backup systems.

Data protection defines constraints on how data is encrypted and stored or transmitted between components of the SoS. Policy statements can specify the types of encryption (RSA, DES, AES, etc.) and constraints over the size of any keys or validity periods [10, 16]. Composing policy statements over a set of interacting components dictates the specific types of data that traverse the SoS. Requirements often coincide with the communication style used by the component for exchanging information (e.g., stream or block transportation).

A final policy type invokes mechanisms used to attain non-repudiation between SoS components. Non-repudiation concerns trust and integrity of information, specifically the cryptographic methods and algorithms that prevent SoS entities from denying having performed actions over data [4]. These policy statements describe the protocols and mechanisms (KE, ZDB, Bao, etc.) that provide non-repudiation [17], as well as any constraints the SoS may place on their configuration such as key sizes and data validity periods. Attributes about the non-repudiation policies include the delivery mechanism used by the algorithm (submission, transport, both) [4] and the interactivity that is required by the system (synchronous, asynchronous) [18, 19].

We do not address policy types that do not directly relate to observable features of SoS components, such as those policies focused on managing incident response [5], environmental protection [5], security user roles [4], and personnel training [5, 20]. While these policies are crucial to the entire certification process and the maintenance of secure systems, the focus of the SCM framework is design-time analysis of the SoS components and their configuration. Thus, a variety of security policy types are ignored including those related to organizational, systems maintenance, managerial, environmental, training, risk assessment, and systems acquisition [5].

1.1 Descriptors

Foremost to formulating a uniform and comparable security policy models is the accumulation of a strong policy information base acquired from reliable sources. The

model should support property and expectation comparisons, not just among policies of the same type, but across policies of different types as dictated by security certification criteria.

Our approach to modeling and comparing the numerous criteria and the qualities embedded in both security policy statements and certification criteria is based on separation of concerns. From this perspective, we introduce eight *descriptors* that structure and organize relevant policy and component information across distinct concerns. There are two classifications of descriptors. *Architectural descriptors* dictate component interaction expectations and partners (neighbors). *Policy descriptors* express the governance properties of each policy. Because they are based on concerns, the segregation of descriptors is independent of the policy type. Therefore, the descriptors span all policy types by allowing overloading of attributes and methods expressed in the resulting profile.

1.1.1 Policy Descriptors

Policy descriptors express concerns related specifically to governing the security of the component. Separating the descriptors across policy information more directly dictates how assessment is targeted. A necessary level of coverage across the information that should be recorded by a component is required to ensure that proper policy description and analysis can occur. Figure 1 shows the interrelation between the five policy descriptors our framework uses for policy description. These are *security policy assertions, observable behaviors, mechanisms, policy constraints,* and *dependencies*. We overview each descriptor model. Definitions of the internal contents of a descriptor can be found in [21].

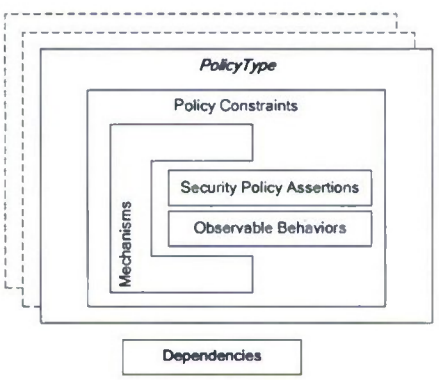

Figure 1. Policy Descriptors

At the core of the description for each policy type are *security policy assertions* and *observable behaviors*. Security policy assertions contain key fundamental statements about the policy statements for each component. The security policy assertions descriptor defines specific policy attributes that are static for each component within the SoS across each policy type (i.e., audit, authorization, etc.). The attributes form a foundation for direct policy comparison to detect conflicts. Assertions values are derived from general policy understanding and common attributes needed for certification. The most basic assertion is the Boolean *capability* that indicates whether or not the component's expose interface has a policy of a particular type. The absence of a policy is a policy in itself and

must be accounted for when assessing compliance [2, 10]. As an example, attributes of the security policy assertions descriptor can detect and ensure that each component contains an audit security policy for compliance assessment of a system-wide policy mandating the centralized collection of audit records from SoS components. Another assertion is *response* type. For example, an audit policy specifies the type of messages that must be generated (e.g., *email message, report, database entry, etc.*) [4].

The *observable behaviors* descriptor uniformly states how a policy restricts component behavior at an exposed interface. Describing the functional aspects of each policy is accomplished via overloaded methods that abstract behavior concepts. The instantiation of this descriptor describes the allowable behavior of the component, as restricted by the security policy, in terms of what information is exchanged between components. Specifying component behavior is a key need for certification. For example, NIST specifies assessment needs with respect to interconnection agreements [22]. The DGSA advises a strict isolation between information domains [3]. The observable behavior descriptor purposely allows overloading its methods to achieve expression across the different policy types. Various interaction behaviors that exist between components include *sendData, receiveData, delegateData, transferData, shareData, initializeData, manageData, broadcastData,* and *requestData,* where *Data* is the data type instantiated in the security policy assertions descriptor for each policy type. For example, *sendData* may refer to the authentication response for an authentication policy or the log file entries for an audit policy. Observable behaviors do not refer to specific code or mechanisms used to transmit data. These concerns are relegated to the mechanism descriptor, described next.

Directly surrounding the core observable behaviors and security policy assertions in the framework is the *mechanism* descriptor that contains references to specific policy enforcement technologies. The mechanism descriptor expresses how a policy is implemented and deployed at an exposed interface. The descriptor records the possible policy enforcement mechanisms that a component supports. Policy enforcement mechanisms have been extensively cataloged by government agencies [9, 15] and certification agencies [2, 4]. The mechanism descriptor encompasses a model the mechanism adheres to and the mode it uses to enforce the policy. It names the data of the correct type given in security policy assertions. Thus, *dataName* indicates an enforcement type such as password, certificates, or PKI [2, 4, 9] for a authentication policy. Another example in which to explain the specific model a policy must adhere to is the case of data protection policies where different encryption algorithms may be used for the encoding of data (RSA, DES, AES, etc.) [10, 16]. The mechanism mode describes details of the policy implementation such as a contingency planning policy that may specify *incremental* versus *differential* backup plans [15, 23].

Dictating inter-descriptor compatibilities within a single policy type is accomplished via the specification of *policy constraints*. Policy constraints selectively apply mechanisms of the framework to observable behaviors. These constraints may encompass combinations of policy attributes, methods, and timing, based on the SoS system requirements and the properties of interaction partners of the component. The policy constraint descriptor supports the differentiation between trusted and untrusted components. If an interaction partner is not known to be trusted, a different set of behaviors are expected that rely on a different set of mechanisms than those for a trusted

component. Thus, the constraints descriptor aids the expression of CERT suggestions for adding additional levels of encryption to communicating components using untrusted network segments [2]. Other types of constraints, such as minimum password lengths [24] and using shared authentication data to develop trust within the SoS [25] are supported via a set of *allow* and *check* methods that rely on UML OCL statements that are expressible within the modeling framework. The constraint descriptor is one of the few descriptors that expects its entries to be customized to the components functionality within the SoS and the neighbors that the component may potentially interact with.

Table 1 provides a sample of the types of overloaded entries that are present in the four descriptors discussed above and shown in the first column. The second column shows sample attributes and methods. The third column indicates possible instantiations. The last column is the policy type for which such an instantiation would be valid. Policies requiring entries not directly supported by the overloaded descriptors can extend the model and define unique policy-specific entries as needed, enabling the framework to uniformly represent a wide variety of policies.

Table 1. Sampling of overloaded descriptor entries

Descriptor	Entry	Example Values	Policy Type
Mechanisms	data	Password, certificate, PKI	Authentication
	data	Symmetric key, public-private pair	Data Protection
	mode	CCB, ECD	Data Protection
	mode	Bit, byte	Non-Repudiation
	model	RBAC, IBAC, MLS, DAC, MAC	Authorization
	model	KM, ZDB, Bao, Markowitch	Non-Repudiation
Observable Behaviors	requestData()	Requests for access	Authorization
	requestData()	Encryption requests for public keys	Data Protection
	response	Allow, deny	Authentication
	response	Allow, deny	Authorization
	shareData()	Transfer log entries for storage	Audit
	shareData()	Copy backup files offsite	Contingency Planning
Security Policy Assertions	dataType	Events, signatures, patterns	Audit
	dataType	Roll back, roll forward	Contingency Planning
	interactivity	Single login, interactive	Authentication
	interactivity	Synchronous, Asynchronous	Non-Repudiation
Policy Constraints	checkDataLength()	Checks length of log files	Audit
	checkDataLength()	Checks length of password	Authentication
	modelComplexity	Password complexity policies	Authentication
	modelComplexity	Encryption algorithm complexity	Data Protection

The last policy descriptor to discuss is the *dependencies* descriptor. This descriptor maintains the desired cross policy constraints for component. For example, authentication policies can depend on audit when a policy statement requires logging the successful or unsuccessful attempts at authentication to detect system intrusion [2, 26]. Dependencies describe interrelationships between different security policy types by explicitly stating links rather than anecdotally via commonly accepted connections. The concern is that if one policy type is in conflict with a neighboring policy type (at an exposed interface) then it is essential to know if that conflict can propagate to other policy types causing an indirect conflict or violation. Thus, the dependencies descriptor expresses inter-policy governance. Dependencies can also be defined from the component's perspective in the SoS to facilitate the analysis of system-wide, cross policy requirements. Multiple descriptor requirements permeate the dependency between policy

types, e.g. (1) two different policy types must both contain security policy assertion descriptors with a true capability, (2) the authentication policy must define observable behaviors that share a Boolean authentication response, (3) the audit policy must support storage of failed authentication attempts, and (4) the audit policy must support a mechanism that delivers alerts to an intrusion detection mechanism. Figure 2 shows a sample of the complex interdependencies between the various policies related to confidentiality [3], secure file backups [2], the protection of trusted functions [4], and other industry standards.

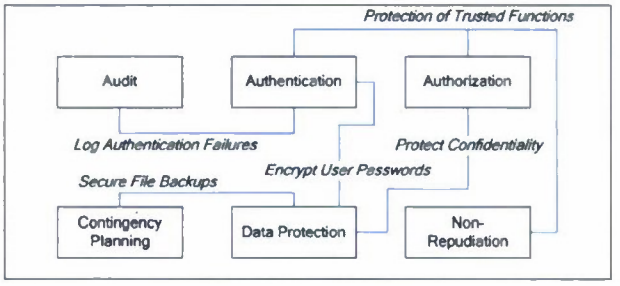

Figure 2. Dependency Descriptor Interrelationship

The combination of the policy descriptors and six policy types along with dependency specifications enables the SCM framework to represent a variety of policy specifications for SoS components. Compliance assessment is possible via analysis of all policy descriptors across all policy types by examining the component linkage that exists within the SoS architecture. Information about SoS links and communication pathways is necessary for the analysis and is defined by a set of architectural descriptors described in the next section.

1.1.2 Architecture Descriptors

In the SCM framework, three *architectural descriptors* express a component's perspective of its processing expectations within the SoS. That is, architecture descriptors indicate who the component's interaction partners are *(configuration)*, how the component communicates to them *(communication)*, what state the component is in *(state)*. By definition of a SoS, each component is linked to one or more interaction partners. The configuration descriptor explicitly identifies the links facilitating the description of trust according to the interaction partners that are under scrutiny. The communication descriptor specifies the architectural characteristics of the link including an interaction style (send, receive, broadcast, poll, etc.) [27, 28], quality, and frequency, as well as the types of data that are sent over the link between components. Communication can be affected by various security policy requirements such as encryption that can reduce the speed and quality of the link [29]. Finally, a state descriptor is used to specify the interaction and data states that a component may encounter as it exchanges data and control with other components. The interactions between components within a SoS can trigger state changes for which security policy specifications dictate behavior. The communication style can be analyzed in conjunction with state and configuration to determine if there are any interoperability problems that can affect security compliance with the components participating in the interaction. The

three can also provide a representation of the SoS overall and its communication infrastructure and composite state.

Table 2 samples some attributes of the architecture descriptors with examples of allowable values that can be used within the CPP.

Table 2. CPP Descriptor Attribute Values

Descriptor Category	Entry	Description	Allowed Arguments / Values
Communication	sendDataOverConnector	Method used to send data to a specific Interaction Partner	Interaction Partner, mech.Data
	receiveDataOverConnector	Method used to receive data from a Interaction Partner	Interaction Partner, mech.Data
Configuration	partners	The set of Interaction Partners associated with a secComp	Interaction Partner

1.2 Certification Criteria

A major difficulty in software security certification is the disconnect between the generic, documented criteria and the formulated requirements statements that are needed to show SoS compliance. The SCM framework we have developed offers a foundation to classify and dissect generic criteria to interpret and express them as requirements with direct applicability to the underlying SoS policy model. The SCM framework then assists the model-based expression of a software component's security policies while simultaneously reformulating criteria statements in a manner that allows evaluation.

The SCM framework has been built to support criteria taken from the DIACAP IA Controls [9], Common Criteria [30], and NIST [15] while still retaining an open approach enabling other criteria documents to be incorporated in the future. For this report, we examine criteria statements taken from the NIST document SP 800-53, summarized in Table 3 [15]. The statements specify the recommended security controls for the auditing of information systems. NIST isolates controls in accordance with security baselines for low-impact, medium-impact, and high-impact information systems. Depending on the classification of a system different security controls are recommended.

Table 3. NIST security controls for audit

Description	NIST 800-53 control no.
The information system generates audit records for events per system *as chosen by the organization* .	AU-2
The information system provides the capability to *compile audit records* from multiple components throughout the system into a *system wide* (logical or physical), time-correlated audit trail.	AU-2(1)
The information system provides the capability to *manage the selection of events* to be audited by *individual components* of the system.	AU-2(2)
The information system produces audit records that contain *sufficient information* to establish what events occurred, the sources of the events, and the outcomes of the events.	AU-3
The information system provides the capability to *centrally manage* the content of audit records generated by individual components throughout the system.	AU-3(2)

As shown in Table 3 each control has one or more topics of focus related to secure auditing. Italics are used to place emphasis on key issues each criteria addresses. The goal is to dissect the criteria, narrow the scope of its software policy applicability, and associate the criteria with the descriptor models. Thus, associating each criteria statement to policy descriptors requires an intelligent decomposition of the criteria. From these associations, directly expressible security requirements emerge with which the modeled SoS must comply. Granted, some criteria statements do not have a direct applicability to the component level. For example, NIST AU-2 stipulates an information system must generate audit records as determined by the *organization*. Certification against this type of criteria statement must be performed at a managerial level in order to determine that the organization has defined the audit record entries the information system must record. No descriptor or policy type in the CPP can adequately prove this statement is satisfied.

Other criteria statements such as AU-2(1) have more direct ties to component policy representations. AU-2(1) contains high-level criteria that "*components must maintain audit records*" and "*components must be able to transfer audit records to other components*" that can be represented with descriptors. Additionally, overlap can exist when examining multiple criteria such as the "*system wide*" audit trail required by AU-2(1) and the "*centrally managed*" audit records stated in AU-3(2).

Our approach uses grammatical constructs and information analysis techniques to form intermediate decompositions for a more general semantic labeling. The intermediate representation can then be mapped into descriptor attributes and methods. Thus, the approach forces the criteria into a requirements statement that is comparable to policy models. Without this uniformity, violations in compliance could not be verified. Table 4 outlines a detailed decomposition of the AU-2(1) criteria mapping specific statements such as "multiple components" to the descriptors for policy constraints and configuration.

Table 4. AU-2(1) Criteria Decomposition and Mapping

Phrase	Intermediate Decomposition	Descriptor Mapping
The information system	System, SoS	System, SoS
provides the capability	Process, function	Mechanism, Policy Constraints
To compile	store, collect, aggregate	Observable Behaviors
audit	policy type	Security Policy Assertions
records	data type	Security Policy Assertions
from	transfer	Communication, Observable Behaviors, Mechanism
multiple	per component	Policy Constraints
components	many components	Configuration
throughout the system	all components possible	Configuration
into	input, receive	Communication, Observable Behaviors, Mechanism
a system-wide	over all components, central	Configuration
time-correlated	data organization type	Security Policy Assertions
audit	policy type	Security Policy Assertions
trail.	storage, series (records)	Security Policy Assertions

Correlating the different descriptor mappings enables more direct codification of criteria details using the CPP model. Each descriptor defines the methods and attributes required for policy expression. The descriptors were refined to further reflect the parameters and arguments as determined by criteria specifications.

1.3 Component Protection Profile

The accumulation of the instantiated descriptors for a particular component forms its *Component Protection Profile* (CPP). The CPP is defined by extending the Unified Modeling Language meta-model (UML). The composition of all CPPs for the components in a SoS represent its set of local policies. Because of the way that we manage the security certification criteria, as discussed above, the CPP also provides the foundational units for criteria expression [31]. Shown in Figure 3 and Figure 4 the CPP distinguishes between architectural and policy descriptors using UML stereotypes to extend the meta-model according to our approach to support security policies. The main component, *secComp,* has associated architecture descriptors, *Configuration* and *State,* that provide a foundation for interaction.

In the lower left of Figure 3, the *Configuration* stereotype contains the set of *Interaction Partners*, each of which is a *secComp* known to be trusted or not by the component. Interaction between components is specified using one or more *Port* specifications. Each *Port* is an exposed interface of the component. The *Port* defines its communication style with the *Communication* descriptor, born out of research results on interoperability analysis and the communication conflicts that can contribute to security vulnerabilities [32, 33]. Enumerations are used throughout the CPP specification to provide the acceptable information for specific types of interaction and policies. For example, an *InteractionType* within the *Communication* descriptor is used by the component to describe the specific type of interaction, e.g., *send, receive, broadcast,* and *poll,* that it expects to exchange data and control with the other components that communicate at that interface port.

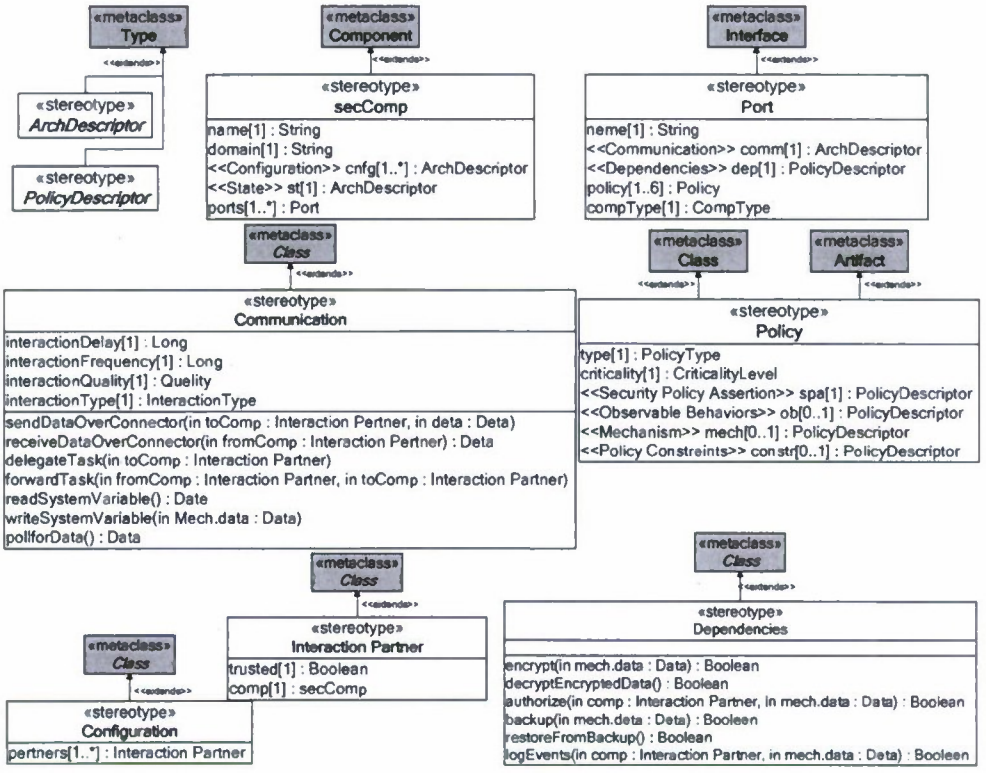

Figure 3: Component Policy Profile and Architectural Descriptors

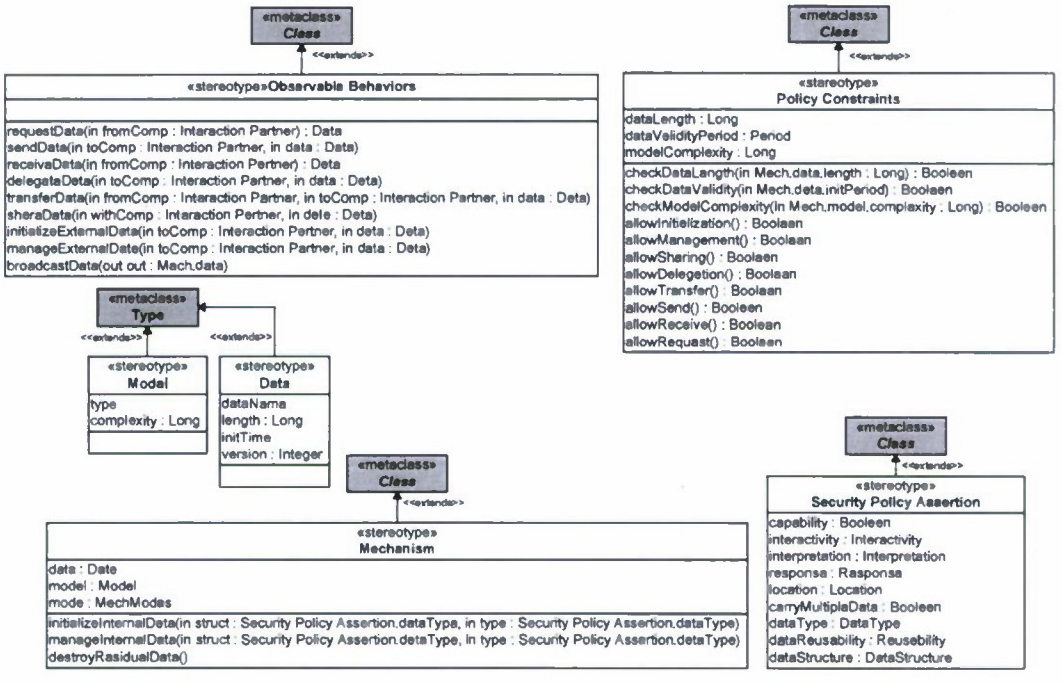

Figure 4. Component Policy Profile Policy Descriptors

Each *Port* can express all six security policy types. This model allows for different policy properties to be expressed depending on what the exposed interface of the component does and whether there is more than one. For example, pulling from a database might be the exchange at one port and pushing data to another component for processing might be at another port. Policy instantiations for SoS components is accomplished by taking the policy specification and mapping it to the appropriate descriptor entries as needed. For example, an audit policy defined for a component necessitates the *Security Policy Assertion* attribute *dataType* to be set to *audit record*.

The CPP also involves the Object Constraint Language (OCL) [34] to establish the formal definitions of the policy constraints and dependencies descriptors as well as safety properties related to certification criteria. Figure 5 below shows an example of the NIST AU-2(1) criteria statement written in OCL and formatted to evaluate over the CPP. Criteria statements at this stage are not dependent on an instantiated CPP, and thus do not require the SoS, its components, or the policies to be known until a compliance check is performed.

Instantiating the CPP for each component represents the sum of the components' security policies. Thus, the SCM framework allows a uniform view of security governance from the component-perspective. Partial specifications of policies are allowed and still contribute to compliance assessment.

Once instantiated CPPS can be reasoned about as composed unit by applying the criteria in conjunction with OCL statements. The SCM framework offers the foundation for a robust representation to determine the presence of incompatibilities and violations of compliance with applicable security certification requirements. Incompatibility in the case of the NIST AU-2(1) is shown by locating a component in the SoS that does not contain an audit policy or that cannot communicate its audit records to another component.

```
context: Policy
inv: Policy.allInstances →                                          // Invariant for all policies in SoS
     ∀ p : Policy | p.name = "Audit"                                // For all policies of type "Audit"
      ⇒
     p.spa.stated ∧ p.spa.dataType = "event" ∧ p.spa.dataStruct = "audit record"   // Policy must be stated
          ∧ p.mech.operations→exists("initializeInternalData")      // initIntData operation must exist
          ∧ p.mech→initializeInternalData(p.spa.dataStruct, p.spa.dataType)   // Policy must use correct mechanism

context: InteractionPartner
inv: InteractionPartner.allInstances →                              // Invariant for all IPs in SoS
     ∃ ip1, ip2 : InteractionPartner | ip1 <> ip2                   // ip1 is not equal to ip2
     ∧ ∃ pol1 : ip1.port.Policy, pol2 : ip2.port.Policy |
          pol1.name = "Audit" AND pol2.name = "Audit"               // Both policies are of type "Audit"
          ∧ pol1.ob.operations→∃ "transferData"
          ∧ pol1.ob→transferData(ip2, pol1.mech.data)
          ∧ ip1.port.comm.operations→∃ "sendDataOverConnector"
          ∧ ip1.port.comm→sendDataOverConnector(ip2, pol1.mech.data)   // ip1 can send data to ip2
          ∧ ip2.port.comm.operations→∃ "receiveDataOverConnector"
          ∧ ip2.port.comm→receiveDataOverConnector(ip1)             // and ip2 can receive data from ip1
```

Figure 5: OCL Constraints on Component Policies for AU-2(1)

The OCL statements in Figure 5 are safety properties – constraints on the policy capabilities. The first AU-2(1) constraint stipulates the requirement for components of the SoS for maintaining audit records. This is stated by requiring that an audit policy

must be stated, that a mechanism in each component must exist to initialize the audit records, and that the component policy must adhere to the correct mechanism for recording audit information. The second constraint of AU-2(1) stipulates that components must support the transfer of audit records to different components. The CPP model uses a concept of *Interaction Partners* to establish communication between components. The OCL constraint states that two components must be able to transmit their respective audit records via connectors within the SoS. Understanding the full meaning of the OCL statements requires the model over which it is defined. These constraints are directly evident within the policy profile of the components and their interaction partners. Thus, verification can performed when the certification criteria are expressed in the same terms as the model.

2. Moving Toward a Formalism

Though the CPP was found to be well structured from a component perspective, the use of OCL for the comparisons against security certification criteria was quite cumbersome (as seen in Figure 5), even for basic safety properties. Therefore, concurrent to exemplifying the model, we investigated the prospect of using a formal specification language. This investigation included a formal definition of the SoS as a composition of component properties, including their policies. However, no languages existed that clearly modeled SoS in a way that would allow us to express security requirements as safety **and** progress properties both locally and globally. Therefore, we first needed to extend a language to accommodate the model constructs. We chose Context Unity [35], a derivative of UNITY [36] as the base formalism because it has a proof theory associated with its execution model. This section discusses the new derivative of UNITY, called X-UNITY (pronounced *Cross*-UNITY), that captures programmatic, structural, and scoping properties of SoSs [37, 38]. We apply similar examples from security audit criteria to illustrate the application of the formalism.

2.1 Some Background in Context Unity

Specification formalisms for SoS must portray hierarchical composition, where intermediate results can be formed and then further composed. We focus on Context UNITY [35] which extends UNITY's programming model [36] to include distribution and interactions with an operational environment through context programs. The primary unit of specification in Context UNITY is the program. Context UNITY represents systems in both an imperative manner (using actual program statements) and a declarative manner (stating program properties). In addition to its specification constructs, it contains an execution model and a proof logic that allows temporal reasoning. We review Context UNITY to the extent that is needed for our extension and examples. Figure 6 shows its basic structure.

Context UNITY program, *P*, describes a state transition system consisting of variable declarations (**declare**), initial values for variables (**initially**), and assignment statements (**assign**). Statements are executed with weak fairness in that they are executed non-deterministically, infinitely often.

```
System SystemName
  Program P<parms>
    declare
      exposed      // public variables
      internal     // private variables
      context      // handles to other public vars
    initially      // initial values for public/private vars
    assign         // programmatic state changes
    context        // programs that use context variables
                   // to interact with environment
  end P
    Components     // the instantiated programs in the system
    Governance     // global impact statements
end SystemName
```

Figure 6: Context UNITY Specification Structure

The **declare** section is divided into variable types for **exposed**, **internal**, and **context**. The **context** program following the **assign** section specifies how changes in the environment's state are reflected in the values of exposed variables, which in turn can influence another program's context variables. Thus, the context program provides components with explicit and individualized interactions within their contexts. The **Components** section is used to define **Program** instances. The **Governance** section contains rules for behaviors that have a global impact on the system. These rules rely on the state of exposed variables throughout the larger system to affect other exposed variables in the system.

Given that programs are actually code, they must be instantiated to form a system. A system may "run" many instances of a program. Program instances in Context UNITY are distinguished by passing parameters (depicted by *<parms>*) during system initialization that includes a unique instance identifier. Thus, Context UNITY provides an initial foundation for structuring the specification of SoS designs.

2.2 Creating X-UNITY

A SoS retains constructs that are derived from its component systems. These restrictions imply that candidate systems for inclusion in the SoS are isolated [38] prior to the SoS being composed. For example, centralized functions may need to be merged across competing/cooperating sub-systems. However, systems which are good candidates for reuse in SoSs have little or no centralized control, facilitating the formation of composites [39]. Therefore, constructs, such as centralized control, are strong indicators of isolation and security compliance failure.

Given SoS characteristics, we must adhere to a specification framework that provides a structure reflecting the concepts of scope, interaction, and reuse while providing mechanisms to support reasoning and proof. The framework requirements should (1) allow multiple architectures for governance and control [39], (2) represent component layering and hierarchies, (3) include imperative and declarative viewpoints, (4) express abstract design and their instantiations, (5) specify different interaction styles (e.g. explicit, implicit, indirect) and (6) depict the concept of reuse of existing program

and system types to model the inheritance of software behaviors from classes or species of software artifacts.

To define X-UNITY

1. We formally induce a hierarchy of module specifications to represent the SoS configuration by augmenting the Context UNITY specification labels of **System** and **Program** with **SoS.**
2. We differentiate explicitly between a system design specification and an instance of the specification as a particular use of the design to reflect the concept of reuse. This differentiation is done with the introduction of **include** for reusing an encapsulated entity and **System Instances** as the instantiation of **Systems.**
3. We allow program variable exposure outside of the scope of the reused system through the introduction of **promote.**

These novel extensions allow X-UNITY to express a SoS so that it can be reasoned about in the context of other systems, not just programs.

Figure 7 shows the basic structure for X-UNITY specifications, illustrating the notation and hierarchy extensions. Though **System** specifications are similar to Context UNITY, our introduction of **include** lets us refer to programs that may be specified elsewhere. Thus, **include** allows the module name to represent its entire specification template. This convention leads to simpler specifications of higher-level systems and provides a consistent notation for reuse using module names. It also makes system composition explicit in X-UNITY, which facilitates reasoning about SoS applications.

A similar convention is introduced at the **SoS** level that encompasses all modular entities. Where a **Program** serves as a template for instances of **Components** in Context UNITY, we extend this approach to include the instantiation of **System** specifications into particular **System Instances**. Thus, when we name a system template as **include System**, this name serves as a reuse symbol in another SoS. When we give an instance of a template a unique identifier as in **Components** or **System Instances**, that instance may be referenced explicitly during execution.

This makes a **System** a type, while a particular **System Instance** is a parameterized occurrence of that type. A **SoS** describes a particular interacting set of system instances. If a SoS is to be reused as a component of another system, it too can be considered a **System** type if needed.

```
System SystemName
  include Program ProgramName1
  include Program ProgramName2
  ...
  Components     // specific program instances
    // ProgramName1(1),..,ProgramName(n)
  Governance
    promote x as w in *

end SystemName

SoS SoSName
  include System SystemName1
  ...
  System Instances
    // configuration of specific system
    // instances within the SoS
end SoSName
```

Figure 7: X-UNITY Basic Specification Structure

The notational extension of **promote** alters the scope of the exposed variable x to include all systems (*). **Promote** makes x available under the alias w in a system s such that $w \in s$.**exposed**; where s.**exposed** is the set of names for all exposed variables available in s. Alterations to the **var** [ ] table in Context UNITY for defining exposed variables are needed to formally represent **promote.** For x's scope to now include s means that a **Program** in s can select the variable x using its attributes, including its aliases, in the **Program context** rules.

Promote appears in the **Governance** section of a **System**. Recall that systems are allowed to have system-wide governance, while SoSs are not. This approach conforms to the definitions of a SoS as a collection of autonomous systems. Thus, **promote** is a core concept for X-UNITY to provide a form of selective composition. It helps capture the unique compositional properties of systems in a SoS. If two systems are formally composed in UNITY, it is done by a union theorem that forces all exposed variables to be public to other systems. The pairing X-UNITY's **promote** with Context UNITY's **uses** rules constrains this formal union to make it possible to achieve more remote interactions with components that are in a local environment. This induces a reaction based on context variables that are quantified over the local environment via **uses**. If such constructions are effective, they create good SoS, even when the elements of the composition are heterogeneous.

2.3 Modeling the SoS Hierarchy

We reuse the security audit criteria from Table 3 within a distributed systems environment as a vehicle for exercising the features of X-UNITY. We take a more global perspective that is rendered in the CPP. We first specify a system of components that capture detected auditable events and generate event notifications within X-UNITY. We limit the code specification to only security audit properties.

Notifications are retained in an exposed variable, *notify*, for later review by auditors. The *notify* variable can be any type of local storage that is accessible to other programs. Here, it is a set of audit records, each of which is an ordered pair with a

timestamp and audit information fields, such as the type of event and the component identifier. The function *detectEvents()* returns the set of events detected in the local environment since it was last invoked. In *AuditableComponent* (Figure 8), once the events are saved in *notify* there are no further state changes.

```
Program AuditableComponent
  declare
    exposed
      notify: Set of AuditRecord
  initially
      notify := ∅
  assign
      notify := notify ∪ detectEvents()
end AuditableComponent
```

Figure 8: The Program *AuditableComponent*

AuditCollector (Figure 9) uses a context variable, *auditCache*, to collect the notifications from components in its same system with the exposed variable, *notify*, where *notify* contains notifications that are not yet in the audit trail. Thus, it does not execute as a stand-alone component. The **uses** statement "loops" over all *p*, in which *n* is local to the scope of the loop. Effectively, component interfaces are advertised by their exposed variables and selected for use by the quantification of **uses** context rules.

In *AuditCollector*, variables named *notify* are selected from all programs *p* that satisfy the **given** clause and are bound to the handle *n*. The "!" notation is used to associate a temporary handle, *n*, to each matched instance of *notify*. **Becomes** is assignment (from Context UNITY). The *auditCache* values are eventually assigned to the exposed variable *auditTrail* using the statement in the program's **assign** section. Weak fairness of UNITY's execution model assures that all statements are selected for execution infinitely often. Given the rules in its **context** program, other components, such as those that instantiate *AuditableComponent*, must provide their audit notifications as exposed variables for collection.

```
Program AuditCollector
  declare
    exposed
      auditTrail: Set of AuditRecord
    context
      auditCache: Set of AuditRecord
  initially
    auditTrail, auditCache := ∅, ∅
  assign
    auditTrail:= auditTrail ∪ auditCache
  context
    auditCache
      uses n!notify in p
      given ¬(n ⊆ auditCache)
      where auditCache becomes auditCache ∪ n
end AuditCollector
```

Figure 9: The Program *AuditCollector*

In Figure 10, we specify a **System** of the components in Figure 8 and Figure 9 using the **include** statement to indicate reuse by copy of the previously defined program types. Without **include**, we would have to repeat the entire program specifications within the system. Reuse results in more complex specifications because all relevant component information must be considered. Therefore, the benefit of **include** is that it provides a construct for better management of these complex system representations, while mimicking actual reuse and composition.

System *CollectedAuditSystem*
  include Program AuditableComponent
  include Program AuditCollector
  Components
    ⟨[] i :: AuditableComponent(i)⟩
    [] AuditCollector
end *CollectedAuditSystem*

Figure 10: The System *CollectedAuditSystem*

A program template (named within an **include** statement) is instantiated in the **Components** section as needed. Instances have unique identities that make them available for later reuse as services by other systems in the **SoS** specification. The notation '[] i ::' means that there are 'i' *AuditableComponents* that execute asynchronously, each with a unique identifier.

To show the compliance of *CollectedAuditSystem* with the requirement of AU-2(1) in Table 3, we formulate a UNITY progress property (P1 below) to generically state that *eventually there is at least one component which has the complete representation of a system-wide audit trail.* This requirement is apparent in the decomposition in Table 4 where the use of the Configuration descriptor is shown. In OCL, the statement can only be that each component has the capability to transfer and there exists a component that can collect them. However, the **leads-to** property is not specifiable in OCL, because OCL can only represent safety constraints. Moreover, the **leads-to** property is global and concise. That is the beauty of the UNITY language. Because we have retained the UNITY execution model, we have use of its temporal proof logic. A "dot" notation expresses the hierarchy of modules and variable names within the X-UNITY specification.

(P1) $\exists c \in C \mid \langle \forall t \in C \mid e \in t.notify$ **leads-to** $e \in c.auditTrail \rangle$

Given the system *CollectedAuditSystem* the statement P1 reasons over all program components ($\forall t \in C$) that are reused within a composite system, such that when t receives an event ($e \in t.notify$) there is at least one component ($\exists c \in C$) that will eventually acquire the event in its audit trail ($e \in c.auditTrail$). Without the X-UNITY extensions, this statement would not be easily expressible or provable, and further reasoning about SoSs would prove difficult. It can be directly seen that *auditTrail* in *AuditCollector* of *CollectedAuditSystem* contains all notifications from all *i,* such that *AuditableComponent(i),* to comply with P1.

2.4 Verifying Compliance of a SoS in X-UNITY

Figure 11 introduces a simple SoS specification as a composition of system specifications, showing X-UNITY's capability to explicitly denote the structural relationship between a SoS and its component systems. In Figure 11, **SoS** *ASReplicas* includes multiple system instances of *CollectedAuditSystem* (Figure 10).

```
SoS ASReplicas
  include System CollectedAuditSystem
  System Instances
    ⟨[] i :: CollectedAuditSystem(i)⟩
end ASReplicas
```

Figure 11: The SoS *ASReplicas*

The constraints of X-UNITY force system-to-system interactions to be made more explicit representing the abstract composition of Configuration descriptors. This convention provides a degree of encapsulation where systems have an inherently defined individuality even if they interact in larger systems. Compositions occur via well defined interfaces at the *Port* (in the CPP) are explicitly exposed and controlled by processes and policies designed for those types of interactions.

In the case of **SoS** *ASReplicas* in Figure 11, each individual *CollectedAuditSystem* can be proven to satisfy P1. Thus, each component complies locally with the audit criteria. However, when reused as multiple instances of a system, **SoS** *ASReplicas* fails to satisfy P1. Each system expects to actively use external information but does not allow access to their internal information. This abstraction would be included in the CPP's mechanism descriptor.

The result is the lack of a SoS-wide audit trail. This occurrence is because each *AuditCollector* within each *CollectedAuditSystem* maintains an audit trail with the records confined to events detected within each component system. This result is a deliberate artifact of modeling systems as autonomous entities that are, by default, closed. However, X-UNITY relies on explicitly declared shared variables to define system-to-system interactions at the SoS specification level.

To overcome the violation so that **SoS** *ASReplicas* complies with P1, we add to Figure 10 a **Governance** section and introduce **promote** to make an exposed variable explicitly visible to other systems. This results in **System** *CollectedAuditSystem-2* as shown in Figure 12.

```
System CollectedAuditSystem-2
  include Program AuditableComponent
  include Program AuditCollector
  Components
    ⟨[] i :: AuditableComponent(i)⟩ [] AuditCollector
  Governance
    promote AuditCollector.auditTrail as notify in *
end CollectedAuditSystem-2
```

Figure 12: System *CollectedAuditSystem-2*

The **promote** statement elevates visibility of specific exposed variables from the **System** level to the **SoS** level. Its use requires clear design intent. Here it is the

certification criteria represented by P1 that guides the choice of the *auditTrail* variable to be renamed as *notify* in order to allow its reference by existing logic in peer systems. This guidance indicates the key influence that the security certification criteria have on model expression, development, and compliance verification. As seen in the earlier sections on the CPP, if the criteria cannot be expressed in the model, then there is no comparative verification that can be directly performed. The investigation into the X-UNITY extension goes further to indicate that the formal expression of security criteria actually guides the model change, if practical, once the non-compliance issue is identified. In this case, it is all peer systems as denoted by '**in** *'. The peer systems are selectable under quantification by **uses** statements in the context programs of peer systems within the SoS.

With this introduction of the **Governance** section and the **promote** statement, every *AuditCollector* component (Figure 9) can access the notifications from each *AuditableComponent* (Figure 8) within its respective system, as well as the top level audit trails of every other copy of *CollectedAuditSystem-2* (Figure 12) within the **SoS** *ASReplicas* (Figure 11). As an alternative to this centralized approach, *notify* in each *AuditableComponent* could have been promoted. However, we choose a centralized design via the **Governance** section because it more closely follows the design approach of making system level interfaces explicit. Promoting *auditTrail* and renaming it to *notify* allows the system to be reused as a component in the hierarchy, while preserving P1 over system quantification.

Note that governance, or control, is either modeled explicitly within a special section of X-UNITY or implicitly through the cooperating code of multiple programs. By definition, it is rare to apply explicit governance to SoS models. Our Context UNITY extensions promote governance rules across the full spectrum of modeling unit granularities (programs, systems and systems-of-systems). For governance to occur explicitly, it must be implemented "outside" the modeling construct (e.g., within the environment) in question. For governance to occur in the SoS, each contributing system must accept some degree of outside control. This control can be in the form of various types of integration middleware that "glue" the systems together. The middleware itself is part of the SoS solution that can be modeled as (1) a subsystem in and of itself or (2) a "governance" function represented explicitly outside of the other SoS parts.

Extending the audit example further, we introduce multiple systems that satisfy the requirements yet do so using different algorithms and system structures. These systems are composed into a larger SoS which may be examined to confirm or deny whether the SoS also satisfies the requirements at the global system level.

```
Program AuditConsumer
  declare
    exposed
      auditTrail: Set of AuditRecord
    context
      auditCache: Set of AuditRecord
  initially
    auditTrail := auditCache := ∅
  assign
    auditTrail := auditTrail ∪ auditCache :
    auditCache := ∅
  context
    auditCache
      uses n!notify in p
      given ¬(n ⊆ auditCache)
      where auditCache becomes auditCache ∪ n
      ∅ impacts n
end AuditConsumer
```

Figure 13: Program *AuditConsumer*

The program *AuditConsumer* (Figure 13) reproduces much of the logic of *AuditCollector* (Figure 9). It differs by 'consuming' the audit events once they are copied to the context variable *auditCache*. The variable *notify* remains the source of audit records in other components throughout the system. Now, *notify* is cleared by an **impacts** statement in the context program. This behavior is captured in Figure 14, *ConsumedAuditSystem*, that instantiates *Auditable-Component* (Figure 8) and the central *AuditConsumer* to gather the audit records for the entire system.

```
System ConsumedAuditSystem
  include Program AuditableComponent
  include Program AuditConsumer
  Components
    ⟨[] i :: AuditableComponent(i)⟩
    [] AuditConsumer
  Governance
    promote AuditConsumer.auditTrail as notify in *
end ConsumedAuditSystem
```

Figure 14: System *ConsumedAuditSystem*

We specify the **SoS** *ConsumeCollectHybrid* (Figure 15) as an SoS of both collecting and consuming audit system types. One collects audit records while leaving their original variables undisturbed, while the other consumes such records and continually clears the source variables. Both report the results as exposed variables named *auditTrail* and promote these variables to peer visibility at the SoS level.

```
SoS ConsumeCollectHybrid
  include System CollectedAuditSystem-2
  include System ConsumedAuditSystem
  System Instances
      ⟨▯ i : 1 ≤ i ≤ N :: CollectedAuditSystem-2(i)⟩
    ▯ ⟨▯ j : 1 ≤ j ≤ M :: ConsumedAuditSystem(j)⟩
end ConsumeCollectHybrid
```

Figure 15: SoS *ConsumeCollectHybrid*

Recall that SoS must satisfy requirements as if they were a single system while not violating properties in their component systems. While the top level audit trail created in the *ConsumedAuditSystem* (Figure 14) satisfies P1, the individual audit trails of its instances in Figure 15 are no longer valid. Their collection algorithm is interfered with by the consumer algorithm within *AuditConsumer* programs in *ConsumedAuditSystem* instances. When the *auditTrail* variables are promoted as *notify*, they too become subject to the **impacts** statements in the context rules of *AuditConsumer* programs and are set to empty sets. This violates P1 for component systems of the SoS. Specifically, instances of *CollectedAuditSystem-2* fail since they no longer have a system-wide audit trail after execution of the **impacts** statement.

The investigation into expressing security certification criteria as progress properties that span the SoS shows two important details. The first is that certification criteria remain in need of a uniform framework for expression type (safety and progress) and designation (local to the component and global to the SoS). Our two models, the CPP and the X-UNITY language, are built on the same foundation of multi-component interaction and behavior expectations. However, one is better at functional representation and the other is better at policy object descriptions. A reconciling of the two along with a methodology for expressing the policies and criteria accurately and completely is still needed for a comprehensive framework.

3. Exploring Access Control Policy Conflicts

Access control policies are defined as a set of individual rules (functions providing privileges) applied to requests from subjects (users) to perform certain actions on a particular set of objects that require a particular access right [40], [41]. Access is granted if the rule evaluation provides privileges for the access requested by the subject. Each component system has its individual access control policy described in terms of hierarchical, separation of duty (SoD), cardinality, and/or time assignments and constraints. The role hierarchy defines seniority among the roles, while SoD constraints restrict access to mutually exclusive operations. Cardinality constraints add numerical restrictions to allowable accesses to a system, and timing, or temporal, constraints define access over a given time interval.

Vulnerabilities related to access control have been defined and organized into different categories that facilitate their detection and resolution [42]. We assume secure components have a domain in which there are no conflicts among their policies. Security vulnerabilities present themselves as policy conflicts or violations that occur due to *inter-domain mappings*, the access control mappings between local component system

domains. Inter-domain access is more specific to the integrated systems where subjects from one component domain try to access objects from another component domain. Inter-domain mappings can also define further restrictions on inter-domain access such as user-role assignments, SoD, role hierarchy, cardinality, and time [43],[44],[45]. With the inter-domain mapping in place, it is possible for access to remain undecided from the integration and for new access decisions to disagree with locally defined policies.

We introduce the Conflict Petri Net (ConPN) to analyze inter-domain access control mappings for SoSs and evaluate their potential to introduce violations to this specific authorization security policy and its internal details. These details are neither local component policy attributes or mechanisms (for representation in the CPP) nor function based for global compliance (for representation in X-UNITY). Instead, they are a third dimension of security certification because they are attached to a policy to govern internal processing that is not available at the exposed interface. Therefore, a different type of analysis is needed to indicate where policy violations may occur.

We show formally how ConPN denotes inter-domain policy violations for Role-Based Access Control (RBAC) systems. Specifically, ConPN examines role inheritance, SoD, cardinality, and temporal policy constraints for compliance. We indicate the violations using their formal definition within ConPN.

3.1 Role Based Access Control and Petri Nets

Role-Based Access Control (RBAC) is commonly used to define access parameters within components and combinations thereof. This definition creates rule-sets of permissions, assigns the rules to roles, and then assigns roles to users [46]. The robust, low-maintenance, and efficient nature of RBAC systems allows for simple modeling of many constraints including hierarchical, SoD, cardinality, and temporal [47]. RBAC systems have noted limitations that should be addressed in integrated system security analysis [48], [43]. Because each user of a system takes on an assigned role, roles should be defined based on how the organization works. Inheritance may be ambiguous when it does not correspond to an organization's hierarchy [48]. Context is not included in role assignment constraints, which can be eased if traceable origins of inherited access are maintained across domains of the integrated system.

A role hierarchy is a partial order relationship established among roles, through which access is granted. SoD constraints define mutually exclusive relations between two entities. Each individual is authorized or not authorized to have access based on the role he or she has been assigned, allowing a system to restrict access to authorized users and manage those permissions associated with groups of users easily by mapping users to roles. Other constraints can be used to restrict access to no more than a specified number of users (cardinality constraints) or to be granted during specific times (temporal constraints).

We graphically represent RBAC systems using the convention where users and roles are nodes in the graph and the connections between them are directed edges, called mappings. The arrow connecting a user to a role represents assignment and the arrow connecting a role to a role represents a hierarchy. Figure 16 illustrates this concept. Role r_1 inherits the permissions given to r_2. Due to the temporal constraint on the mapping between r_1 and r_2, r_1 only inherits from r_2 on Monday, Tuesday, and Wednesday. Because r_2 inherits privileges from r_3, r_1 also indirectly inherits permissions from r_3. Role r_3 does

not inherit from any other roles. User 1 is given all permissions defined for roles r_2 and r_3 because of the user-role assignment to r_2 and the role hierarchy between r_2 and r_3. The cardinality constraint on r_3 indicates that only one user is allowed to have r_3's privileges at a time. This means that if another user is assigned to any of the other roles, due to inheritance, this constraint would be violated. The mapping between roles r_3 and r_4 is considered an inter-domain mapping because it spans from one domain to another.

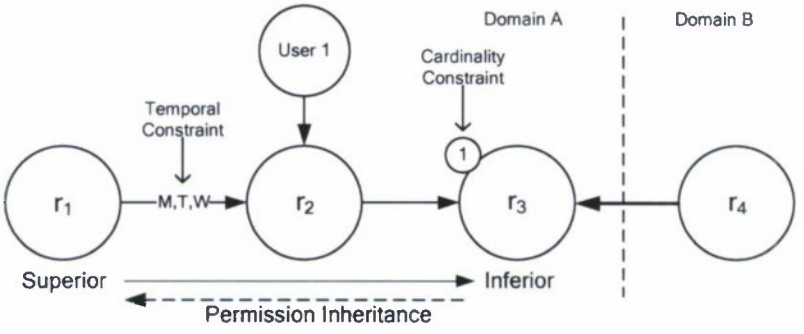

Figure 16. Role Hierarchy and Permission Inheritance

Petri Nets describe systems at various levels of abstraction, and when combined with the ability to represent hierarchies, modeling complex systems becomes much easier [49]. Petri Nets are bipartite directed graphs, making it easy for modeling and formal verification. The theory behind Petri Nets allows flexibility to extend existing models once they conform to the basic Petri Net constraints. Moreover, they can capture both static and dynamic aspects of a system, which is not possible in other techniques like graph-based models.

A Petri Net is a graph, $G_{PN} = (V, E)$, where the set, V, of vertices is comprised of *places* and *transitions* and E is the set of edges, or *arcs* between them. A place is never connected to another place directly, and transitions are never connected to another transition directly. Places are static entities. Transitions represent dynamic entities because the transition firing rules can change the contents of the tokens that flow through the Petri Net. A Petri Net is a 3-tuple:

PN = (P, T, F) such that
P: Set of *places*, $P \subseteq V$
T: Set of *transitions*, $T \subseteq V$, $P \cap T = \varnothing$
F: Flow relation for *arcs*, $F = (P \times T) \cup (T \times P)$, $F \subseteq E$

We define the following specific Petri Net entities

- *Input Arc*: Flow f represents an input arc for transition tr when f = (p, tr) such that $f \in F$, $p \in P$, and $tr \in T$ because it flows from a place into a transition.
- *Output Arc*: Flow f represents an output arc for transition tr when f = (tr, p) such that $f \in F$, $p \in P$, and $tr \in T$ because it flows from a transition to a place.
- *Input Place*: $p \in P$ is an input place when it is connected to a transition $tr \in T$ through an input arc.

- *Output Place*: $p \in P$ is an output place when it is connected to a transition $tr \in T$ through an output arc.

There can be multiple input and output places for a given transition, which then forms a set of input places and a set of output places. In the most basic transition, tokens flow from all input places into all output places, even if the number of input and output places differs. Graphically, a place is represented by a circle; a transition by a rectangle or a bar; and an arc with an arrow (Figure 17).

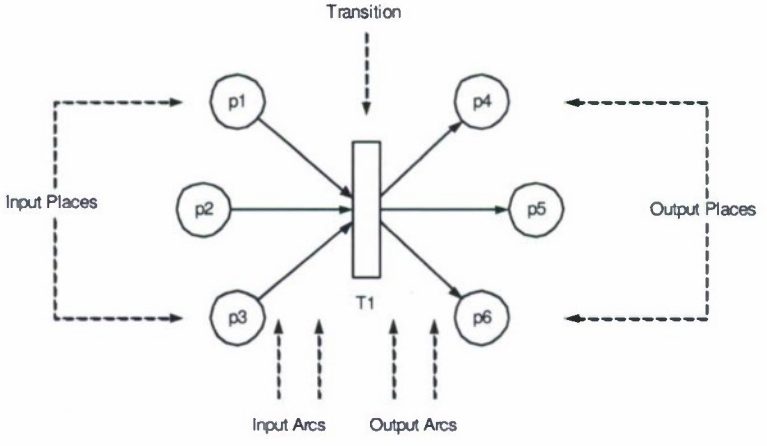

Figure 17. Basic Petri Net

- *Token*: A token is the entity that flows within an executing Petri Net, represented by small dots inside a place.
- *Enabled Transition*: A transition becomes enabled when all its input places have at least one token.
- *Fired Transition*: An enabled transition is fired (Figure 18) upon removing the tokens from the input place and placing them into the output places according to the firing rules.

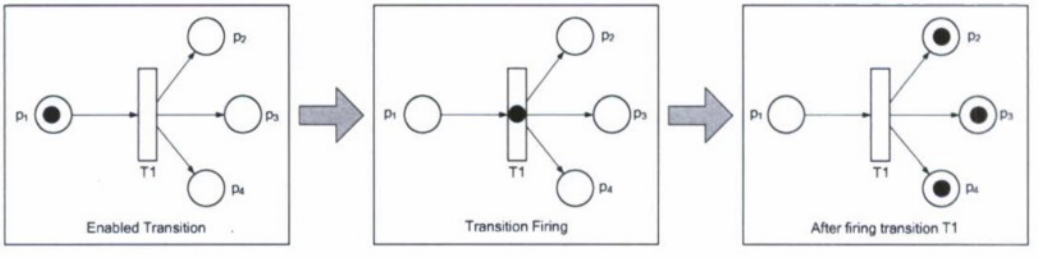

Figure 18. Transition Firing

Executing a Petri Net is moving a set of tokens through the graph via transition firing rules. A Petri Net *executes* while it has enabled transitions that can be fired. This is known as a *liveness* property of a Petri Net. When a Petri Net reaches a state where no transition can be fired, it is known to be *dead*. Colored Petri Nets allow distinguishable tokens by assigning a particular color to a token [46]. They also introduce the concept of

arc-expressions, binding variables, and guards. These improvements dramatically increase the overall expressive power of Petri Nets by bringing them closer to programming languages.

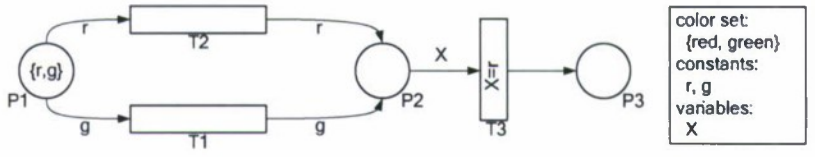

Figure 19. Colored Petri Net

In Figure 19 we can see how a *color set* is defined. In our ConPN, we use this concept to place necessary information within tokens so that conflicts can be found. Here, place P1 can hold red or green tokens. The arc between P1 and T1 allows only green tokens and the arc between P1 and T2 allows only red tokens. In this case, the token color represents a data type, and each place, transition, or arc can place requirements as to what color/type of token can exist on it. Place P2 does not put any restriction on what color the tokens need to be, and the variable X can be either red or green. The transition T3 has a guard expression and is known as a *guard transition.* Depending upon the guard evaluation, the transition will fire or not fire. This means that if X is red, then T3 will fire, putting the token into place P3. Similarly, arcs also can have arc expressions whose functionality is the same as guard expressions.

3.3 Inheritance Policy Conflicts

We define an inheritance policy as a 4-tuple, *InheritancePolicy(U, R, SoD, M)* such that

U: Finite set of Users

R: Finite set of Roles

SoD: Finite set of role based SoD requirements expressed as a triple (u, r_1, r_2) where $u \in U$ and $\{r_1, r_2\} \subseteq R$

M: Finite set of user-to-role and role-to-role mappings (or assignments) with cardinality constraints expressed as triples (u, r_1, n) or (r_1, r_2, n) where $u \in U$, $\{r_1, r_2\} \subseteq R$, and $n \in \aleph$.

A role in *R* can have a restricted cardinality or may allow infinitely many users in *U* to have access. A *SoD* requires at least two roles in R to indicate that the same user in *U* cannot be involved in both at the same time. If a mapping in *M* has a temporal constraint, then the inheritance of a role is restricted to a certain time interval as expressed by a natural number. If the is no temporal constraint, then $n = \infty$.

Let IP_A and IP_B be the inheritance policies of domains A and B, respectively, which may represent independent components. Let IP_{join} be the inter-domain mappings that tie the access policies together.

Let $IP = IP_A \cup IP_B \cup IP_{join}$ be an inheritance policy. An *inheritance conflict* exists when a role inherits permissions it should not be allowed to have. This generally occurs with an incorrect role hierarchy between superior and inferior roles

Conflicts of interest exist when entities (users or roles) should not be instantiated at the same time and are either directly or indirectly allowed to do so because of faulty

inter-domain mappings. Conflicts of interest are often forcibly prevented by including a SoD constraint on roles. If SoD lines are incorrectly mapped, when two domains are combined into one system, a role may be accessed by two conflicting users at the same time [50].

Cardinality and temporal constraints extend the conflict of interest constraints with time and assignment number restrictions. Cardinality constraints assign an upper limit to the number of users assigned to a role at one time. Temporal constraints assign certain time-periods for which a mapping is valid. A temporal conflict arises if a particular user can be assigned to a specific role that does not have equivalent temporal units. If this is the case, then either the user can access the role at a time that is not permitted or the role is incorrectly unavailable to the user.

We have augmented an example initially published in [43] to demonstrate the conflicts defined above and how they are depicted in ConPN. Figure 20 depicts access control policies of Domain A, Domain B, and the inter-domain mapping between A and B.

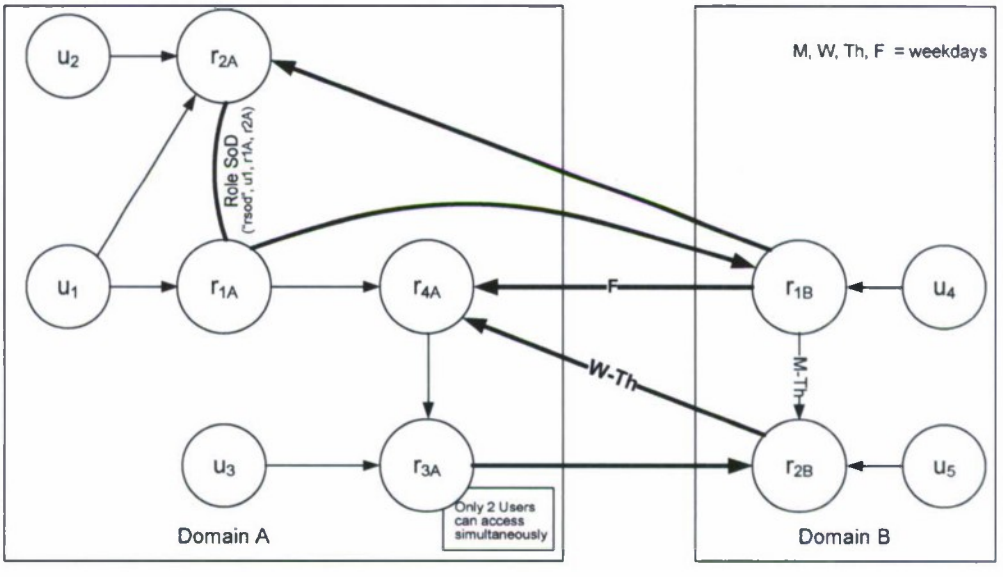

Figure 20. Motivating Example

Thus, $IP = IP_A \cup IP_B \cup IP_{join}$ where

$IP_A = (U_A, R_A, SoD_A, M_A)$, such that
$U_A = \{ u_1, u_2, u_3\}$,
$R_A = \{ r_{1A}, r_{2A}, r_{3A}, r_{4A}\}$,
$SoD_A = \{(u_1, r_{1A}, r_{2A})\}$,
$M_A = \{ (u_1,r_{1A},\infty), (u_1,r_{2A},\infty), (u_2,r_{2A},\infty), (u_3,r_{3A},\infty), (r_{1A},r_{4A},\infty), (r_{4A},r_{3A},\infty)\}$
$IP_B = (U_B, R_B, SoD_B, M_B)$, such that
$U_B = \{u_4, u_5 \}$,
$R_B = \{r_{1B}, r_{2B} \}$,
$SoD_B = \{\}$,
$M_B = \{(u_4,r_{1B},\infty), (u_5,r_{2B},\infty), (r_{1B},r_{2B},\text{M-Th})\}$
$IP_{join} = (U_{join}, R_{join}, SoD_{join}, M_{join})$, such that
$U_{join} = \{\}$,
$R_{join} = \{\}$,
$SoD_{join} = \{\}$,
$M_{join} = \{(r_{1B},r_{2A},\infty), (r_{1A},r_{1B},\infty), (r_{1B},r_{4A},F), (r_{2B},r_{4A},\text{W-Th}), (r_{3A},r_{2B},\infty)\}$

The inter-domain mapping, denoted by M_{join}, induces all five types of conflicts to occur. We show how ConPN can detect the potential for these conflicts. Using a Petri Net allows us to separate concerns among conflict types and form our model atop a commonly accepted formal technique. This process is an improvement over manually scanning graphs or XML documents to find conflicts, such as in [43] where only conflict resolution is automated.

Access control conflicts can allow the most unqualified user access to the most sensitive information if even one mapping is incorrectly specified. Depending on the sensitivity of the system, this could allow information leaks that affect a company's survival against competition or it could threaten national security by allowing attackers into sensitive government systems. The goal of ConPN is to guarantee that all policy conflicts are found, which is the only way to definitively say that a system's access control is secure. Thus, the approach will err on the side of false positives, rather than missing any potential conflict. Realizing the consequences of improper access, the operation of ConPN is designed to monitor a user's access path to indicate what roles a user has been given access to. This eases conflict resolution as it helps isolate where in the Inheritance Policy faulty mappings have occurred.

3.5 Constructing the Conflict Petri Net

We build upon the foundation of Colored Petri Nets to create the Conflict Petri Net, *ConPN*. ConPN requires extensions to the definitions of places, tokens, and arcs without interfering with the basic rules of execution and analysis of the Colored Petri Net. We formally define the transition rules that underlying the perspective-based, conflict detection mechanism. The structure of ConPN represents role-based access control policies easily and completely.

ConPN is a graph, $D = (P, A)$, where vertices are comprised of start places, role places, and choice places and edges are comprised of input and output arcs. Tokens show the policy in motion by flowing from place to place using transitions and firing rules to

traverse the arcs. ConPN retains meta-data based on token movement and the roles visited to determine if a state can be reached that indicates a policy violation.

Places. A place is defined as follows:
Place = (*ID*, *CurrTk*, *TkLog*) such that

ID:	Unique place identifier
CurrTk:	Current token set at place
TkLog:	Bag of tokens that have visited the place, can be null

The set of all places, P, in ConPN is partitioned into Normal and Choice places. Normal places are further partitioned into Start places (*SP*) and Role places (*RP*). The set of Choice places (*CP*) simulate the controlled access to certain roles for the specified user as dictated by SoD requirements. Choice places differ from start and role places because they embody these essential policy constraints. Thus,

$$SP \subseteq P \wedge RP \subseteq P \wedge CP \subseteq P \wedge (SP \cap RP \cap CP = \varnothing)$$

All users have a 1:1 mapping with Start places in ConPN, signifying user access to the system. All roles in *IP* have a 1:1 mapping with Role places. SoD constraint triples have a 1:1 mapping with Choice places. For example, the inheritance policy represented by Figure 20 translates into five Start places with *ID*s $\{u_1, u_2, u_3, u_4, u_5\}$, seven Role places with *ID*s $\{r_{1A}, r_{2A}, r_{3A}, r_{4A}, r_{1B}, r_{2B}\}$, and one Choice place, $\{rsod_1\}$.

The cardinality of each place in P is determined by the function C: $P \rightarrow \aleph$, which maps a place to a natural number indicating the maximum number of tokens allowed to flow to that place. If no cardinality restriction for some place $p \in P$ exists, then $C(p) = \infty$. In our example, only one role, r_{3A}, in Figure 20 has a cardinality restriction, such that $C(r_{3A}) = 2$.

Figure 21 shows the organization of the places in ConPN that correspond to the example in Figure 20.

Tokens. In ConPN, tokens represent the execution semantics of the inheritance policy. Tokens facilitate snapshot and post execution analyses that identify inter-domain policy conflicts. Their values can be updated and evaluated throughout the ConPN execution. Similar to tokens in a Colored Petri Net, tokens in ConPN are distinguishable from one another. Let *TK* be the set of all possible tokens allowed in the ConPN. We define a *Token* as follows.

Token = (*Type*, *Origin*, *Time*, *RLog*, *ConflictID*) such that

Type:	normal or choice
Origin:	Origin of this token, start or choice place
Time:	Temporal Unit, initially empty
RLog:	Bag of roles, initially empty
ConflictID:	Set of place IDs

There are two types of tokens, normal and choice. The origin of the token indicates where the token starts, i.e., a start or choice place. Every start place has exactly one normal token that originates there (recall the 1:1 mapping of user to start place). Every choice place has exactly one choice token that originates there (recall the 1:1

mapping of SoD requirements to choice places). Figure 21 shows the initial token placements for the example. The token time is the current temporal unit held by the token as seen in Table 5 and is initialized to the empty set. As normal tokens move through the ConPN, they collect the IDs of the roles visited in *RLog*. This is a bag because normal tokens can visit the same role multiple times and each visit is recorded. *RLog* is initially empty. Choice tokens work to model SoD conflicts. Since these token types are restricted in their movements, they use *ConflictID* to initialize the role places that cannot be jointly accessed. *ConflictID* is always empty for normal tokens.

Since start places and choice places in ConPN are each initialized with a single independent token of the proper type. When a start or role place has multiple output arcs, it replicates its token for each output arc. An example of this is role r_{1B} in Figure 21 in which a token from r_{1a} or u_4 will be replicated. In contrast, a choice place does not replicate tokens.

Table 5: Initial Tokens and Places

Tokens	Places
tk_1 = ("normal",u_1,∅, ∅, ∅)	u_1
tk_2 = ("normal",u_2,∅, ∅, ∅)	u_2
tk_3 = ("normal",u_3,∅, ∅, ∅)	u_3
tk_4 = ("normal",u_4,∅, ∅, ∅)	u_4
tk_5 = ("normal",u_5,∅, ∅, ∅)	u_5
tk_6 =("choice",$rsod_1$,∅, ∅, {r_{1A},r_{2A}})	$rsod_1$

Transitions. As defined for the basic Petri Net, places are not directly connected to other places. Tokens must flow through a transition in the set *T*, from an input place to an output place according to transition firing rules. In ConPN, we extend the concept of a transition to include a function Temporal: $T \rightarrow TU$, which maps a transition to a set of temporal units from the power set of all such units, TU. These temporal units indicate the constraints on the time at which the transition's input place can pass a token to its output place.

Each role assignment in the inheritance policy (visualized by a directional arrow in Figure 20) has a corresponding transition, directly mapping to 14 transitions. We add to the set those transitions supporting the Role SoD. For the each Role SoD triple in the inheritance policy (u_1, r_{1A}, r_{2A}), we include a new transition to each choice place that branches to those transitions associated with the competing role assignments. Thus, from our example, we introduce transition tr_{15} between r_{1A} and $rsod_1$. The final set of ConPN transitions generated by the inheritance policy in Figure 20 is $T = \{tr_1, \ldots, tr_{15}\}$.

Arcs. Recall that an input arc connects a place in *P* to a transition in *T* and an output arc is directed from transition to place. Let *Arc* be the set of all arcs in the ConPN. To create the notion of SoD that disallows access to a role because access has been granted to a competing role, we provide each arc with a status by defining Status: $Arc \rightarrow$ {"active", "blocked"} to indicate if an arc is valid to transition a token from one place to the next.

Figure 21 shows the complete ConPN generated from the same inheritance policy as the example in Figure 20.

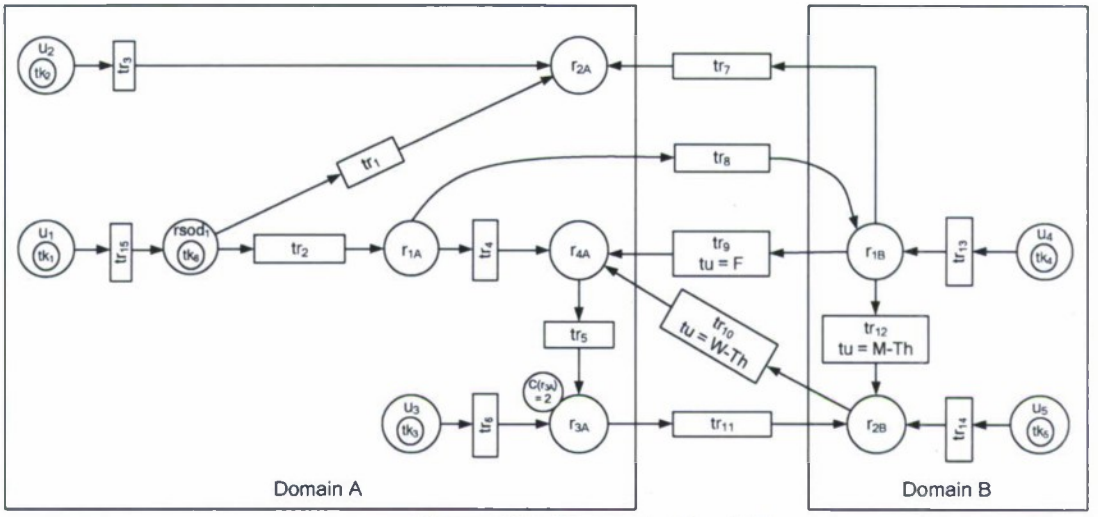

Figure 21. Example ConPN

Transition Firing Rules. As the ConPN executes, a *transition firing rule* analyzes its places, tokens, and arcs to dictate the flow of the tokens throughout the net. Tokens can flow in parallel. We tailor the transition rules so that the movements of tokens through the ConPN imitate the access granted through IP_A, IP_B, and IP_{Join}. The transition firing rules reflect the Inheritance, SoD, Cardinality, and Temporal constraints.

We use a precondition/postcondition format to express the transition firing rules. Notationally, ":=" is pseudo code for *gets*, "/" represents *set delete*, and "Θ" represents *bag union*. Only changes to entities are detailed in the postconditions, whose statement order is meaningful.

Transition firing rule **TR1** moves a normal token (*tk*) from a normal or start input place (p_1) to the transition's (*tr*) output place (p_2) that may be of any type. Token and place meta-data (*CurrTK*, *RLog*, and *TkLog*) are updated. If the transition has defined temporal restrictions, the temporal units of the normal token (*Time*) take on the intersection of the time units associated with the transition as dictated by the security principle [40]. Arc status is unaffected by **TR1**. We formally define the conditions under which **TR1** fires as follows.

For $tr \in T$; $tk \in TK$; $arc_1, arc_2 \in Arc$; $p_1 \in SP \cup RP$; $p_2 \in P$

Preconditions:

$arc_1 = (p_1, tr) \wedge arc_2 = (tr, p_2) \wedge \text{Status}(arc_1) = \text{Status}(arc_2) = \text{"active"}$

$tk \in p_1.CurrTk \wedge tk.Type = \text{"normal"}$

Postconditions:

$p_1.\text{CurrTk} := p_1.CurrTk / \{tk\}$

$p_2.CurrTk := p_2.CurrTk \Theta \{tk\}$

$p_2.TkLog := p_2.TkLog \Theta \{tk\}$

$tk.Time := \text{tk.time} \cap \text{Temporal}(tr)$

$tk.RLog := tk.RLog \Theta \{p_2\}$

This transition firing rule, **TR1**, executes when there are active arcs connecting two places with a normal token being moved between them. The place (p_1) containing the token is either a start place (*SP*) or a role place (*RP*). The place where the token moves (p_2) can be any place within the ConPN. After the transition fires, the p_1 deletes the token from its set of current tokens (p_1.*CurrTk*). The token's time constraint (*tk.Time*) is updated to match the temporal units of the transition being fired. The token also updates its log of visited roles (*tk.RLog*) to include p_2 and the token is added to p_2's set of current tokens (p_2.*CurrTk*). Finally, the token log for p_2 is updated to include the token (p_2.*TkLog*). Figure 22 depicts the resulting changes in the ConPN when **TR1** is applied to transition tr_9.

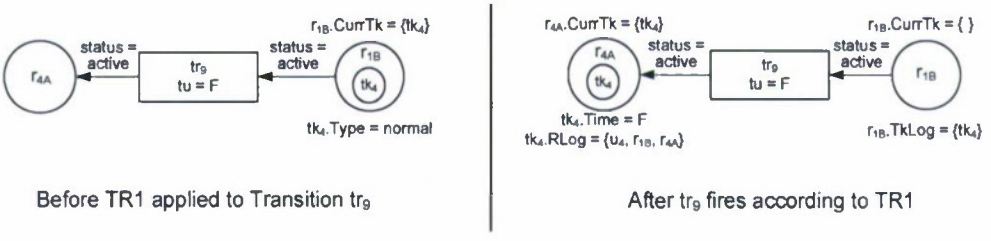

Figure 22. Transition Firing Rule TR1 as Applied to Transition tr_9

Transition firing rules **TR2** and **TR3** both rely on a choice place as their input place to enforce the restriction implicit in a role-based SoD requirement. **TR2** dictates the firing of the transition to move a normal token (*tk*) to a chosen role (*r*) when it resides at the choice place (*c*) where a choice token (*ctk*) also resides.

The choice token must be forced to move to the alternate role where it stays for the duration of the execution. The enforcement occurs because **TR2** changes the state of output arc (arc_1) that the normal token uses to "blocked." The choice token then has only one active arc to leave the choice place. **TR3** dictates the firing of the choice token (*ctk*), relying on the firing of **TR2** as indicated by the presence of a blocked arc (arc_1). For a conflict to exist there must be a secondary path to the non-chosen role. Hence, if the same normal token arrives at the place where the choice token newly resides, then the SoD requirement is violated.

TR2 fires under the following conditions.

For $tr \in T$; $tk, ctk \in Token$; $arc_1, arc_2 \in Arc$; $c \in RSoD$; $r \in RP$

Preconditions:

$arc_1 = (c,tr) \wedge arc_2 = (tr,r) \wedge$ Status(arc_1) = Status(arc_2) = "active"

$\{tk, ctk\} \subseteq c.CurrTk$

$tk.Type$ = "normal" $\wedge$ $ctk.Type$ = "choice" $\wedge$ $ctk.Origin = c$

Postconditions:

$c.CurrTk := c.CurrTk / \{tk\}$

status(arc_1) := "blocked"

$r.CurrTk := r.CurrTk \Theta \{tk\}$

$r.TkLog := r.TkLog \Theta \{tk\}$

$tk.Time : tk.Time \wedge$Temporal(tr)

$tk.RLog := tk.RLog \Theta \{r\}$

Transition Firing Rule **TR2** applies when there are two active arcs (arc_1, arc_2) connecting a choice place (c) to a role place (r), where the choice place contains both a normal token (tk) and a choice token (ctk). The choice token originates ($ctk.Origin$) at the choice place where it is residing. After **TR2** fires, the choice place removes the normal token (tk) from its set of current tokens ($c.CurrTk$). The token adopts the temporal constraints of the transition ($tk.Time$), the token's log of visited roles ($tk.RLog$) now includes the role place (r), and the token is added to the role place's set of current tokens ($r.CurrTk$) and its log of tokens ($r.TkLog$). This rule additionally sets the status of the arc between the choice place and the transition (arc_1) as *blocked*. The choice token's existence is necessary to determining if **TR2** is enabled, but the choice token is moved separately using **TR3**, described next. **TR3** fires under the following conditions.

For $tr_1, tr_2 \in T$; $ctk \in Token$; $arc_1, arc_2, arc_3 \in Arc$; $c \in CP$; $r \in RP$

Preconditions:

$arc_1 = (c, tr_1) \wedge arc_2 = (c, tr_2) \wedge arc_3 = (tr_2, r)$

Status(arc_1) = "blocked"

Status(arc_2) = Status(arc_3) = "active"

$ctk \in c.CurrTk \wedge ctk.Type$ = "choice" $\wedge$ $ctk.Origin = c$

Postconditions:

$c.CurrTk := c.CurrTk / \{ctk\}$

$ctk.ConflictID := ctk.ConflictID / \{r\}$

$r.CurrTk := r.CurrTk \Theta \{ctk\}$

$r.TkLog := r.TkLog \Theta \{ctk\}$

The choice token is moved by **TR3**, which by definition can only be applied after **TR2** is fired. This is because of the precondition that an output arc (arc_1) from the choice place (c) is blocked. The choice token (ctk) must both originate from and currently reside in the choice place. Once **TR3** fires, the choice token is removed from the choice place ($c.CurrTk$) and the set of conflicting id's ($ctk.ConflictID$) is changed to omit the role place (r) where the choice token now resides. Only the alternate role is left in the

ConflictID set. There are no transition firing rules to move the choice token. Thus, if a normal token joins this choice token and the normal token has visited the role in the choice token's ConflictID set, then the SoD requirement has been violated because the normal token has visited both roles places and should not have been able to. The role place (*r*) then adds the choice token to its set of current tokens (*r.CurrTk*) and its token log (*r.TkLog*).

Since choice tokens are not transitioned further by any firing rules and are not included in inheritance, cardinality, or temporal conflict assessment, no further postcondition changes are warranted than those presented above. It is important to note that the finding of a single conflict denotes a problem with this SoD constraint. Therefore, it is not important to the conflict detection that the arc blocks further flow from the RSoD place to the chosen role.

Figure 23 illustrates these transition firing rules on the ConPN example shown in Figure 21.

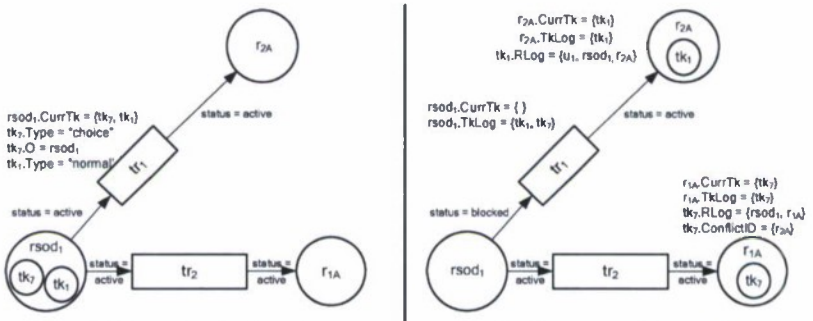

Figure 23. Transition Firing Rules TR2 and TR3

These transition firing rules provide the means to detect conflicts in the ConPN. By moving tokens through the Petri net, different states of the system are found. The ConPN executes in a step-wise fashion where enabled transitions are fired until a quiescent state is reached. This organized evaluation is easily automated using ConPN-specific software [12]. The next section shows how inheritance policy violations are found from the ConPN execution.

3.6 Finding Conflicts with ConPN

The ConPN can detect inheritance, role-based SoD, cardinality, and temporal compliance violations when representing an inheritance policy, *IP*. We discuss these findings below.

Inheritance Conflict. If a token visits the same role multiple times, it means that a role in *IP* can be visited by a cyclical firing of transitions in the ConPN. This state is evident in the *RLog* of each token that retains the roles visited by the token. Thus, the

cycle is visible if any place, other than a choice place, appears in the *RLog* of a normal token more than once. Formally,

$\exists\, tk \in Token; r \in SP \cup RP$ such that $tk.RLog(r) > 1 \Leftrightarrow$ InheritanceConflict(r)

We use a formal definition of the bag, *RLog*, as a function from places to the set of natural numbers indicating the number of times a role appears in the bag. Thus, $tk.RLog(r)$ returns the number of times r appears in $tk.RLog$.

Figure 24 shows a potential inheritance conflict in the ConPN using the example from Figure 21. Since $tk_3.RLog(r_{3A}) > 1$, it is clear that a violation exists. This violation occurs because in IP_{join} $\{(r_{2B},r_{4A},$W-Th$), (r_{3A},r_{2B},\infty)\} \subseteq M_{join}$. This mapping translates into an execution of the ConPN in which the normal token, tk_3, inside role r_{3A} is shown as having progressed through the sequence of transitions $\langle tr_6, tr_{11}, tr_{10}, tr_3\rangle$ to progress from u_3 to r_{3A}, r_{3A} to r_{2B}, r_{2B} to r_{4A}, and back to r_{3A}. As shown in bold text in Figure 24, the tk_3.RLog contains role r_{3A} twice indicating an inheritance conflict. Because r_{3A} is both superior and inferior to r_{4A}, the RBAC system cannot determine which role is genuinely superior.

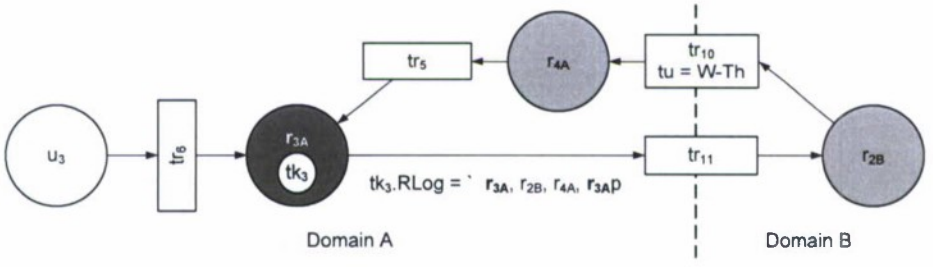

Figure 24. Inheritance Conflict

SoD Conflict. A potential role-based SoD conflict is detected if a normal token visits a role place that holds a choice token under the following conditions.

- The choice token is not at its place of origin.
- The choice token holds in its ConflictID set a role that the normal token has visited.

This indicates that the choice token was unable to protect the use of the role, where it currently resides, from being accessed by a token that has already accessed the competing role governed by the RSoD constraint. This is formalized as follows.

$\exists tk, ctk \in Token; r_1, r_2 \in RP$ such that
$tk.type =$ "normal" $\wedge$ $ctk.type =$ "choice" $\wedge$
$\{tk, ctk\} \subseteq r_1.CurrTk \wedge ctk.Origin \neq r_1 \wedge$
$r_2 \in tk.RLog \wedge r_2 \in ctk.ConflictID$
$\Leftrightarrow RSoDConflict(r_1)$

For role-based SoD transitions, the transition firing rules **TR2** and **TR3** are the only ones enabled. Each role-based SoD place holds a choice token, where its set of conflicting IDs (*ConflictID*) is initialized with the competing roles. Upon the first normal token's path being taken from a SoD place, **TR2** is performed. The normal token moves to the next place and the arc it took becomes blocked. **TR3** becomes enabled in this state. When **TR3** fires, it sends the residing choice token down the alternate path from the normal token. Choice tokens will never be moved from a role place since no transition firing rules exist to move them out of a role place.

Figure 25 shows the role-based SoD conflict in the example. The lightly shaded role places (r_{1B}, r_{1A}) indicate the path of the normal token. The more darkly shaded place (r_{2A}) indicates the conflict where the two tokens arrive at the same place. User u_1 initially contains the normal token tk_1. The choice place, $rsod_1$, initially contains the choice token tk_6. When these tokens are both contained in $rsod_1$, they are moved according to rules **TR2** and **TR3**. Assume the case in which tk_1 moves through tr_2 to r_{1A}. By **TR3**, tk_6 is moved to r_{2A} through tr_1. The *ConflictID* set of tk_6 becomes $\{r_{1A}\}$. As tk_1 moves through the ConPN, it is able to access r_{2A}. A violation occurs because its RLog contains r_{1A}. Thus, user u_1 can be assigned roles r_{1A} and r_{2A} simultaneously, bypassing the Role SoD constraint between roles r_{1A} and r_{2A}.

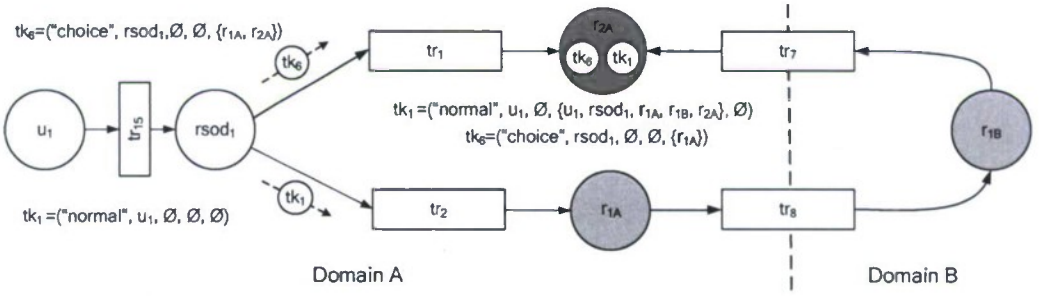

Figure 25. Role-Based SoD Conflict

Cardinality Conflict. A potential cardinality conflict is present in the ConPN when more tokens visit a role place than its cardinality allows. Cardinality conflicts are formally defined as follows, with the symbol '#' meaning 'size of'.

$\exists r \in RP$ such that $C(r) < \#r.TkLog \Leftrightarrow CardinalityConflict(r)$

Figure 26 shows the cardinality conflict in the ConPN where tokens t_1, t_3, and t_4 have reached role place r_{3A}. According to IP_A, u_1 and u_3 can legitimately access r_{3A}. Given that $(r_{1B}, r_{4A}, F) \in M_{join}$, user u_4 can also access role r_{3A} leading to cardinality conflict as the number of tokens reaching role place r_{3A} exceeds the cardinality limit. Though, not shown in Figure 26, u_5 can also access r_{3A}.

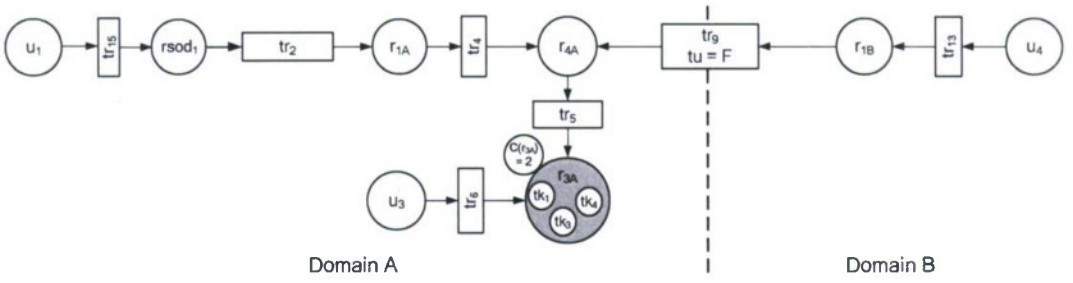

Figure 26. Cardinality Conflict

Temporal Conflict. A potential temporal conflict is present if tokens originating from the same start place can reach the same role place via distinct paths and end up with unequal temporal units. The inequality means that the role has been over or under restricted due to the inter-domain mappings. Both cases are considered improper access. Formally, a potential temporal conflict exists when

$\exists tk_1, tk_2 \in Token; r_1 \in SP; r_2 \in RP$ such that
$(tk_1.Origin = r_1 \wedge tk^2.Origin = r_1) \wedge$
$r_2 \in tk_1.RLog \wedge r_2 \in tk_2.RLog \wedge tk_1.RLog \neq tk_2.RLog) \wedge$
$(tk_1.Time \neq tk_2.Time)$
$\Leftrightarrow$ TemporalConflict(r_1, r_2)

The temporal unit of an inter-domain mapping is the smallest unit granularity on which temporal constraints are defined, for example days or hours. Temporal constraints are dictated by a transition when the token's *Time* is assigned the value of the Temporal function for the transition. Figure 27 illustrates how the ConPN identifies this conflict type. User u_4 has access to r_{4A} via two paths: $u_4 \rightarrow r_{1B} \rightarrow r_{4A}$ and $u_4 \rightarrow r_{1B} \rightarrow r_{2B} \rightarrow r_{4A}$. Note that r_{1B} replicates token tk_4 to create tk_4'. Transitions tr_9 and tr_{10} have different temporal units, which cause confusion as to when u_4 should be allowed r_{4A}'s permissions. Because the two temporal units are different, improper access can occur. This state occurs even each transition reassigns the temporal units to the intersection of the existing Time value and the transition's time constraint.

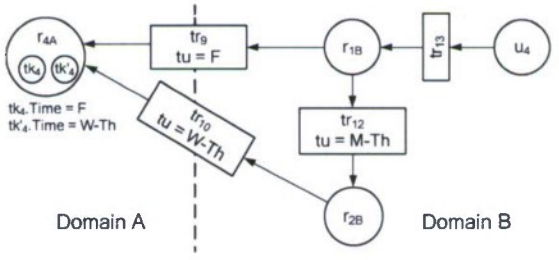

Figure 27. Temporal Conflict

Each component in a SoS has its own access control domain. Certain interactions among the software components within a SoS imply the need to configure an inter-domain mappings of access privileges. These mappings require formal, scalable scrutiny

to uncover potential violations of confidentiality and availability within a single component. ConPN offers a solution to some of the intricate analysis to discover potential violations.

4. Conclusion

The difficult integration process of composing a set of independent software components can introduce unexpected security vulnerabilities in a SoS, making security certification difficult. We have found that multiple dimensions of policy expression are needed to detect and mitigate vulnerabilities. In this report, we highlight three dimensions and their policy representations, analyses, and verification strategies.

5. References

[1] 44 U.S.C. Statue 3542, "44 U.S.C. Chapter 35 - Coordination of Federation Information Policy - Subchapter III - Information Security," 2007.

[2] J. H. Allen, *CERT Guide to Systems and Netwrok Security Practices*, 2 ed.: Addison Wesley, 2001.

[3] E. A. Feustel and T. Mayfield, "The DGSA: Unmet Information Security Challenges for Operating System Designers," *ACM SIGOPS Operating Systems Review,* vol. 32, pp. 3-22, 1998.

[4] A. McCullagh and W. Caelli, "Non-Repudiation in the Digital Environment ". vol. 5, 2000.

[5] National Institute of Standards and Technology (NIST), "FIPS Publication 200: Minimum Security Requirements for Federal Information and Information Systems," 2006.

[6] National Institute of Standards and Technology (NIST), "Special Publications (800 Series)," 2009, http://csrc.nist.gov/publications/PubsSPs.html.

[7] "Auditing and logging of activities in a network," in *IT-Grundschutz Manual.*

[8] M. Swanson and B. Guttman, "Generally Accepted Principles and Practices for Securing Information Technology Systems," National Institute of Standards and Technology, Ed., 1996.

[9] Department of Defense, "DoD Instruction 8500-2: Department of Defense Information Assurance Certification and Accreditation Process (DIACAP)," 2003.

[10] C. P. Pfleeger, *Security in Comupting*, 4th ed.: Prenteice-Hall, 2008.

[11] M. Smith, "Toward Policy Based Logic For Secure Component Interaction," in *Department of Computer Science*. vol. M.S. Tulsa: University of Tulsa, 2006.

[12] A. Walvekar, "ConPN: Detecting Conflicts in Interdomain Mappings," in *M.S. Thesis, Department of Mathematical and Computer Sciences* Tulsa, OK: University of Tulsa, 2006.

[13] M. Lorch, S. Proctor, R. Lepro, D. Kafura, and S. Shah, "First experiences using XACML for access control in distributed systems," in *2003 ACM workshop on XML security*, Fairfax, VA, 2003.

[14] M. Swanson, A. Wohl, L. Pope, T. Grace, J. Hash, and R. Thomas, "Contingency Planning Guide for Information Technology Systems," National Institute of Standard ad Technology, Ed., 2002.

[15] National Institute of Standards and Technology (NIST), "Recommended Security Controls for Federal Information Systems, NIST Special Publication 800-53, Revision 2," U.S. Department of Commerce, Ed., 2007.
[16] National Institute of Standards and Technology (NIST), "Guideline for Implementing Cryptography in the Federal Government, NIST Special Publication 800-21, Revision 2," U.S. Department of Commerce, Ed., 2008.
[17] S. Kremer, O. Markowitch, and J. Zhou, "An Intensive Survey of Fair Non-Repudiation Protocols " *Computer Communications,* vol. 25, pp. 1606--1621, 2002.
[18] P. Louridas, "Some guidelines for non-repudiation protocols," *ACM SIGCOMM Computer Communication Review* vol. 30, pp. 29-38, October 2000.
[19] S. Gurgens, C. Rudolph, and H. Vogt, "On the security of fair non-repudiation protocols," *International Journal of Information Security,* vol. 4, pp. 253 - 262 October 2005.
[20] Office of Management and Budget, "Appendix III to OMB Circular No. A-130," 2008.
[21] M. Kelkar, "Modeling Software Component Security Policies," in *Department of Computer Science.* vol. Ph.D. Tulsa: University of Tulsa, 2007.
[22] R. Ross, "Guide for the Security Certification and Accreditation of Fedaral Information Systems," May 2004.
[23] National Institute of Standards and Technology (NIST), "Contingency Planning Guide for Information Technology Systems, NIST Special Publication 800-34," U.S. Department of Commerce, Ed., 2002.
[24] National Institute of Standards and Technology (NIST), "Electronic Authentication Guideline, NIST Special Publication 800-63," U.S. Department of Commerce, Ed., 2006.
[25] B. Sotomayor, "The Globus Toolkit 3 Programmer;s Tutorial," 2004, http://gdp.globus.org/gt3-tutorial/singlehtml/progtutorial_0.4.3.html#id2514656
[26] M. A. Kelkar, R. Perry, R. Gamble, and A. Walvekar, "The Impact of Certification Criteria on Integrated COTS Based Systems," in *sixth IEEE International Conference on Systems Composition and Interoperability*, Banff, Alberta, Canada, 2007.
[27] L. Davis, R. Gamble, M. Hepner, and M. A. Kelkar, "Toward Formalizing Service Integration Glue Code," in *IEEE International Conference on Service Computing*, Orlando, FL, 2005.
[28] N. Medvidovic, "On the Role of Middleware in Architecture-Based Software Development," in *14th Int'l SEKE*, Ischia, Italy, 2002.
[29] M. Anderson, "VoIP Security: Uncovered," 2005.
[30] International Organization for Standardization (ISO), "Common Criteria for Information Technology Security Evaluation - Part 2: Security Functional Components," in *Version 3.1 Revision 2, CCMB-2007-09-002*, 2007.
[31] Object Management Group (OMG), "UML 2.1.2 Superstructure and Infrastructure," 2007, http://www.omg.org/spec/UML/2.1.2/.
[32] L. Davis, R. Gamble, J. Payton, G. Jónsdóttir, and D. Underwood, "A Notation for Problematic Architecture Interactions," in *ACM SIGSOFT's Symposium on the Foundations of Software Engineering* Vienna, Austria, 2001.

[33] J. Payton, G. Jónsdóttir, D. Flagg, and R. Gamble, "Merging Integration Solutions for Architecture and Security Mismatch," in *International Conference on COTS-Based Software Systems*, Orlando FL, 2002.

[34] Object Management Group (OMG), "Object Constraint Language Specification, version 2.0," 2005, http://www.omg.org/technology/documents/modeling_spec_catalog.htm#OCL.

[35] G.-C. Roman, C. Julien, and J. Payton, "Modeling adaptive behaviors in context UNITY," *Theoretical Computer Science,* vol. 376, pp. 185-204, 2007.

[36] K. M. Chandy and J. Misra, *Parallel Program Design: A Foundation*: Addison-Wesley, 1988.

[37] M. T. Gamble and R. Gamble, "Reasoning about Hybrid Systems of Systems Deisgns," in *IEEE International Conference on Composition Based Software Systems*, 2008.

[38] M. T. Gamble and R. Gamble, "Isolation in Design Reuse," *Journal of Software Process Improvement and Practice,* vol. 13, pp. 145-156, 2008.

[39] E. Morris, P. Place, and D. Smith, "System-of-Systems Governance: New Patterns of Thought," Carnegie Mellon University, Technical Note CMU/SEI-2006-TN-036, 2006.

[40] M. Bishop, *Computer Security: Art and Science*: Addison-Wesley, 2003.

[41] S. Goel, C. Clifton, and A. Rosenthal, "Derived access control specification for XML," in *ACM workshop on XML security* Fairfax, Virginia, 2003, pp. 1 - 14

[42] M. Kelkar, M. Smith, and R. Gamble, "Interaction Partnering Criteria for COTS Components," in *18th International Conference on Software Engineering and Knowledge Engineering (SEKE'06)*, 2006.

[43] R. Bhatti, B. Shafiq, E. Bertino, A. Ghafoor, and J. B. Joshi, "X-GTRBAC Admin: A Decentralized Administration Model for Enterprise-Wide Access Control," *ACM Transactions on Information and System Security,* vol. 8, pp. 388-423, November 2005.

[44] I. Ray, R. France, N. Li, and G. Georg, "An Aspect-Based Approach to Modeling Access Control Concerns," *Information and Software Technology,* vol. 40, pp. 557 - 633, April 25 2004.

[45] B. D. Joshi, B. Shafiq, A. Ghafoor, and B. Bertino, "Dependencies and Separation of Duty Constraints in GTRBAC," in *eighth ACM symposium on Access control models and technologies*, Como, Italy, 2003, pp. 51-64.

[46] K. Jensen, *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use* vol. 1: Springer-Verlag, 1992.

[47] B. Shafiq, J. B. Joshi, E. Bertino, and A. Ghafoor, "Secure Interoperation in a Multidomain Environment Employing RBAC Policies," *IEEE Transactions on Knowledge and Data Engineering,* vol. 17, pp. 1557-1577, Nov 2005.

[48] A. A. El Kalam, R. E. Baida, P. Balbiani, S. Benferhat, F. Cuppens, Y. Deswarte, A. Miege, C. Saurel, and G. Trouessin, "Organization based access control," in *Proceedings of the 4th International Workship on Policies for Distributed Systems and Networks (POLICY '03)*, Toulouse, France, 2003, pp. 120- 131.

[49] W. Reisig, *Petri Nets: An Introduction*: Springer-Verlag, 1985.

[50] D. F. Ferraiolo, G. Ahn, R. Chandramouli, and S. I. Gavrila, "The Role Control Center: Features and Case Studies," in *ACM Symposium on Access Control Models and Technologies* Villa Gallia, Como, Italy 2003, pp. 12-20.

[51] M. Smith and R. Gamble, "Equating ConPN with Inheritance Policies," University of Tulsa, Technical Report SEAT-UTULSA-2007-4, 2007.